The Northern Bites 2008 Standard Platform Robot Team

Prof. Eric Chown
Jeremy Fishman, Johannes Strom, George Slavov
Tucker Hermans, Nicholas Dunn, Andrew Lawrence
John Morrison, Elise Krob
Department of Computer Science
Bowdoin College
8650 College Station
Brunswick, ME, 04011-8486, USA
jgmorris@bowdoin.edu
echown@bowdoin.edu
http://robocup.bowdoin.edu

November 11, 2008

Abstract

This document serves as the Technical Report for The Northern Bites RoboCup team, which participated in RoboCup 2008 in Suzhou, China.

Contents

1 Introduction **1**
1.1 RoboCup 2009 . . . 1
1.2 Team Information . . . 1

2 Vision **2**
2.1 AIBO Vision . . . 2
2.2 Color . . . 3
2.3 Ball . . . 4
2.4 Goals . . . 5
2.4.1 Looking for posts . . . 5
2.4.2 Identifying posts . . . 6
2.4.3 Finding secondary posts . . . 8
2.4.4 Finding backstops . . . 8
2.5 Beacons . . . 11
2.6 Field Line/Corner Overhaul . . . 11
2.6.1 High level overview . . . 12
2.6.2 Line point detection . . . 12
2.6.3 Create lines from points . . . 13
2.6.4 Join line segments . . . 13

2.6.5 Fit unused points . . . 14
2.6.6 Extend lines . . . 14
2.6.7 Intersect Lines . . . 15
2.6.8 Center circle checks . . . 16
2.6.9 Identify corners . . . 16
2.7 Transition from AIBO to Nao . . . 16
2.7.1 Robot Detection . . . 17
2.7.2 Camera Hack . . . 17
2.7.3 Pose-based distance estimation and horizon line . . . 17

3 Motion **18**
3.1 Forward Kinematics . . . 18
3.2 Inverse Kinematics . . . 19
3.3 Aldebaran Motion Parameters . . . 19

4 Behaviors **20**
4.1 Finite State Automaton . . . 20
4.1.1 Player . . . 21
4.1.2 Tracking . . . 21
4.1.3 Navigation . . . 22
4.2 Coordinated Behaviors . . . 22
4.2.1 Strategies . . . 22
4.2.2 Formations . . . 22
4.2.3 Roles . . . 23
4.2.4 Sub-Roles . . . 24
4.2.5 Role Switching . . . 24
4.2.6 System Breakdown . . . 25
4.3 Playbook Editor . . . 25

5 Current and Future Work **27**
5.1 Vision . . . 27
5.2 Motion System . . . 27
5.3 Localization . . . 27
5.4 Behaviors . . . 28

6 Code Diff **28**

7 Code Availability **28**

1 Introduction

Northern Bites is committed to implementing its own solutions to the problems faced in RoboCup. Last year, on the AIBO, we worked to refine our behaviors and team strategy. The Nao presented us with an opportunity to rework our code base and apply a new perspective to some of the classic RoboCup problems.

1.1 RoboCup 2009

Bowdoin College's Standard Platform RoboCup Team, the Northern Bites, intends to participate in the 2009 RoboCup World Championships. As a small, liberal-arts, undergraduate college, Bowdoin is not a typical entry in RoboCup. We have shown over the last few years, though, that we can still contribute significantly to the league, and compete with the best of teams. Encouraged by our strong performances with the AIBO platform, we are confident and enthusiastic about working with the Naos. This year's team has a strong leadership contingent with a trio of captains and a growing young base of new members.

1.2 Team Information

Professor Eric Chown founded the Northern Bites in the spring of 2005. The team grew out of a fall 2004 *CSCI 320: Robotics* course. In its infancy, the team had only one student member who worked on it as a senior honors thesis. That version of the team competed at the 2005 U.S. Open. What was unusual about the team was that all of the code was written by only one student. None of the code came from other teams.

The 2006 team introduced a new code base and bigger development team. Seven students took independent, accredited studies working on the development for the 2006 team and were joined by eight or so more members who contributed in their spare time. A significant result of that year's team was that it tied for 10th place at RoboCup 2006 in the Northern Bites' first year in international competition. Also that year's team started a weblog to document and discuss their progress of our team, accessible at: `http://robocup.bowdoin.edu/blog/`.

The team improved every aspect of the code base in time for the German Open in April 2007 and finished in third place. Significant improvements were made in localization and high level strategy. Development continued at a fervent pace over the summer in preparation for RoboCup 2007. We continued to hone and develop our Python based behaviors, as well as making significant improvements in lower level systems, such as vision and motion. Our changes helped pave the way for a first place finish in Atlanta at RoboCup 2007.

Last year, we continued to hone our high level strategies on the AIBO as we learned about the new Nao platform. With the AIBO, we adjusted our strategies to accommodate five robots and worked to refine our teamwork. For the Naos, we ported much of our AIBO code, such as vision and localization, to the new platform. We also developed a new high level behavior structure.

RoboCup is an important part of Computer Science here at Bowdoin. After our victory, we have seen interest in the department and the team increase significantly. We have attracted new members to our team who otherwise might not have considered a career in computer science. RoboCup is also important to Bowdoin because of the undergraduate research opportunities it provides. This year, three team members are working on RoboCup related senior honors theses. As a small school

with a relatively small Computer Science department, RoboCup gives students the chance to apply computer science in a hands on, team-oriented research and development team.

2 Vision

This year we have made a number of important changes to our vision system, including a complete restructuring of our field line and corner detection. It was reworked to be much more extensible and sophisticated, to replace a patchwork system put together over the past few years. Basic robot detection was also added to help improve our behaviors. The increase in the size of the field presented some problems for the small image size, so there was work done to increase the accuracy of recognition of distant objects. We also ported all of our vision code onto the new Nao platform and adapted it to run quickly on a larger image size.

This section of the paper will first describe our vision algorithm on the AIBO, including detection of beacons, goals, backstops and lines. The final subsection will explain how we adapted our vision processing to run on the Nao.

2.1 AIBO Vision

In 2007 the Northern Bites began to move away from the methods traditionally favored by teams in the four-legged league – notably a reliance on run-length encoding and blobbing. While these techniques have not completely disappeared in our code they have only limited importance in how we process vision. This change is a result of our desire to move towards a more cognitive approach as well as our attempt to meet the challenges involved in recognizing the new goals introduced in 2007.

For most teams in RoboCup vision proceeds in a number of steps. During the first step the image is scanned pixel by pixel (most teams do this horizontally, but we do it vertically from the bottom of the image up). Each pixel is categorized as either being a specific color or it is marked as undefined. Connected pixels are grouped together by a process known as run-length encoding This gives the vision system blobs of specific colors that it can then use as the basis of object recognition. The great advantage of this method is its simplicity and that it works well with the sorts of objects that make up the RoboCup environment (field objects are basically rectangles). On the downside it is not terribly efficient. It is often the case, for example, that significant chunks of the visual field do not contain useful information. Since our team leader's interest lies in the human spatial system we have chosen to use this as an entry to exploring a more cognitive visual system. For example, people typically only attend to a few items at a time in the visual field and are adept at ignoring vast amounts of visual information.

With this in mind we wrote an experimental vision system during the Fall 2006 based upon more cognitive ideas. The idea was to use knowledge about the field in conjunction with limited short-term memory to guide the search for field objects. The approach had a lot in common with The basic outline of the approach went as follows: first we used guided search to extract the boundaries of the field. This can be done quickly by using simple scans that look for groups of green pixels. It can be sped up by using memory – e.g. where the boundary was last frame – and by using pose information. Once the shape of the field has been extracted the next step is to search for field objects (goals and beacons). Again the search can be guided by a combination of the field boundary, memory, and constraints provided by field geometry. Additional information

can also be brought to bear, e.g. beacons interrupt the line that defines the edge of the field. Once a potential object was found it was not processed as a blob, but it was actively scanned to check for its shape. For example a potential beacon might be identified by a scanline of white-blue-yellow. The actual object recognition was done while the pixels were being scanned for the first time. After this the interior of the field can be processed. A working prototype of this system was created that showed a great deal of promise - it recognized objects as accurately as our old system and processed significantly fewer pixels in doing so.

At this point we were faced with the choice of modifying our old system to deal with the new goals or continuing development on the new, unproven, system. In addition we were not certain at that time how well our new system would cope with rotated images. Ultimately we decided on a hybrid system that contained pieces of each. Based upon our experience this year, next year we hope to move completely away from a run-length encoding system.

Our current system begins by processing field shape information. This information is used mostly to constrain object recognition. We then proceed by scanning the image vertically collecting runs of connected colored pixels just as if we were going to do run-length encoding. However, we use these runs of pixels mainly as a guide to search for field objects. Once we have identified candidate objects we do an active scan that attempts to match the expected shape against what is in the image. The process breaks down as follows:

1. Find the edges of the field if any are in the visual field.
2. Scan the image from bottom to top collecting runs of individual colors. During this process we also look for specific color transitions (e.g. blue to yellow, yellow to blue).
3. Look for beacons. These can be found only at locations where the color transitions occurred.
4. Look for goals. The search for goals can be constrained by the presence of beacons. Normally there will be a substantial separation in the visual field between beacons and goals. The heuristic we use to guide the search is to start with the biggest run of color (either blue or yellow) that occurs near a field edge (but isn't too close to a beacon). Once we find a candidate goal we first try to identify a single post using an active scan (described below). Next we attempt to identify which post it is. Then we look for its mate.
5. Look for the ball. This is one case where we still default to simple blobbing.
6. Identify lines and corner.

2.2 Color

Like most teams in the four-legged league our vision system begins by categorizing individual pixels as colors. Following the NUbots we use two types of colors: 1) "Hard" colors, and 2) "Soft" colors. Hard colors are pure versions of colors such as Green or Blue while soft colors combine two hard colors, e.g. BlueGreen. We have found that the necessity of various soft colors is dependent upon the camera settings that we use. For example, when using "Indoor" settings we find that the Blue and Green colors are poorly separated. On the other hand when using "Fluorescent" the problems are more with the separation of Yellow and White. In competition we use the Indoor setting.

Color classification is performed using a simple lookup table. As with other teams we do not use all of the information in the pixel values which would result in a table of size 256x256x256 and would

not afford any generalization in the calibration process. Instead we throw one bit of information away from each dimension resulting in a table size of 128x128x128. We have experimented with first doing some calibration on an even more compact table (e.g. 64x128x128), then resizing the table and doing fine tuning on the larger table, but we have so far found it to be more trouble than it is worth.

The table is built by a calibration process. This can be done by streaming images directly from the robot or by using snapshots taken by robots. In competition where wireless is spotty we generally rely on using snapshots. Our calibration system is built around the idea of image segmentation. We start by performing a simple segmentation on the image we are using. This segmentation looks for large variations in values of adjacent pixels (the variation amount can be set on the fly by the person doing the calibration). This variation can be in either Y, U or V. Once the image is segmented the person doing the calibration simply picks a color and clicks on the image. The system then classifies the selected pixel according to the current color and recursively expands the area until either a segmented pixel is detected or we hit some preset limit (also set by the person doing the calibration – we generally use distances of less than ten to the original pixel). When a pixel classification is at odds with a previous classification (e.g. it was Blue, but now is Green) it is eligible to become a Soft color. We do not do this for all combinations of colors, just for BlueGreen, YellowWhite, RedOrange, and OrangeYellow.

Because the bulk of our object identification is done without blobbing we use Soft colors differently from other teams. When we "scan" an object the Soft color is allowed to substitute for the color we are looking for. For example, when looking for a Blue goalpost we treat BlueGreen pixels as being Blue. However, when we are doing sanity checks on objects we normally treat these pixels as not being the right color. So, for example, we would reject a goalpost if it consisted solely of BlueGreen pixels or even if the percentage of BlueGreen pixels was too high.

2.3 Ball

The ball is the one field object where we mainly rely on blobbing. Our algorithm involves first collecting up candidate balls by blobbing. During this process we treat Soft colors with Orange (e.g. OrangeRed) as being Orange. We then proceed by examining our candidates one by one, from the largest to the smallest, until we find one that passes all of our filtering tests. Since the ball is so important in RoboCup the screening process is quite rigorous.

We start by differentiating between large and small balls as small blobs are much more likely to generate false positives. For example one filter involves the percentage of Orange pixels in the blob. For small blobs we are not tolerant of Soft pixels, but for larger blobs we will treat them as being Orange as long as the total percentage of OrangeRed is not too high. Our filters generally fall into three categories:

1. **Where is the ball?** One form of this is that the ball should either be directly on the field or it should be at the field horizon. Blobs found above the field horizon are ignored regardless of their size. We also throw balls away depending on where they are in the visual field relative to the field horizon. For example, if the field horizon is high in the visual field (meaning that we are looking at a relatively downward angle) it should not be possible to see small balls near the bottom of the image.

2. **How round is the ball?** Again we check this more carefully on small balls than on large balls. The problem with the roundness check is that many balls are partially occluded and

therefore are not particularly round. This can occur when the ball is at one of the edges of the visual field, or because another robot is partially obscuring it. As part of this process we try and determine if the ball is indeed occluded and if so what is occluding it. For example if the ball is being occluded in the upper right quadrant by something White it may be the case that another robot is trapping it. This process essentially consists of finding the center of the blob and scanning out in each of the eight cardinal directions.

3. **What is surrounding the ball?** As usual this varies for large and small balls. For small balls we want to see evidence that we are not looking at a Red uniform. Among the things that serve as positive evidence are the presence of Green pixels and a lack of Red pixels in the area around the blob.

Once we have a candidate ball we still need to determine its distance. We can do this one of two ways. In the ideal case we do a simple calculation based upon the size of the blob. However, when the ball is partially occluded this can be inaccurate. In such cases we try to fit the blob to a circle using standard circle fitting techniques Once we have extracted the circle it is a simple matter to calculate the distance using normal methods.

2.4 Goals

The presence of new goals was the main impetus for an almost complete rewrite of our vision system. A blobbing system worked well with the old goals because they consisted of a single rectangle of color. With the new goals, on the other hand, the goals can consist of anywhere from one to four or more rectangles depending on the angle and occlusion. A blobbing approach simply will not capture the structure involved. In addition, one of the goals in seeing a goal is to determine where to shoot the ball. With the old goals robots could simply shoot at the goal color. With the new goals this is not a good strategy as the color will mainly be seen in the posts. Instead we would like to shoot between the posts. Therefore the primary aim of our new goal recognition system is to identify where the posts are. Our goal recognition system consists of a number of distinct steps:

1. **Find the largest post.** We use a simple but effective heuristic to do this. Our algorithm starts by grabbing the longest run of color found during our vertical color segmentation scans.
2. **Identify which post we are looking at.**
3. **Look for the post's mate.**
4. **Look for the "backstop" between the posts.**

We now look at each of these steps in turn.

2.4.1 Looking for posts

We start with a strip of color. Our goal is to build the largest rectangle that contains the strip. Among the difficulties are the fact that posts are attached to crossbars and backstops. The posts may also be partially occluded. Our algorithm is built around two crucial routines: **verticalScan** and **horizontalScan**. Both of these routines take in an X, Y position as a starting point and will do an axis-parallel scan (where the defining axis is the horizon) stopping when the scan finds

enough pixels which are of the wrong color. For example, if we are looking for the blue goal we will scan for Blue or BlueGreen pixels but count pixels along the way that do not match. When the number reaches some threshold we stop, marking the last place we saw a matching pixel.

In principle the algorithm is simple. Starting at some point X, Y we scan up and down to get the top and bottom boundaries of the rectangle. Then we scan left and right from the midpoint. This gets the right and left boundaries. At this point it is just a matter of doing some additional scans to ensure the quality of the boundaries. In practice matters are not quite so simple. For example, the scans out from the midpoint could contain the backstop or crossbar (Fig. 1). For this reason we perform an additional set of scans to ensure that each vertical or horizontal strip of the rectangle contains a sufficient percentage of pixels of the correct color. This effectively shrinks the rectangle into a core that is solidly colored.



Figure 1: **Rectangle Extraction** Sometimes the backstop extends from the middle of the post.

At this point we do some basic filtering to make sure it is a worthy perspective goal. Basically we check that it is large enough, the ratio of height to width is within tolerance, and that its relationship to the field horizon is reasonable. If it passes these tests then we try and identify which of the two sides of the goal it is on.

2.4.2 Identifying posts

The structure of the goal makes post identification surprisingly difficult. For example, an obvious algorithm is to scan out from the sides of a post looking for crossbars or backstops. However, when viewing the post from the side this algorithm would produce incorrect labels (Fig. 2). Another difficulty comes when the post is partially occluded. In such a case the rectangle defining the post will rarely be ideal. For these reasons we have a host of methods ranging from the highly accurate to what amounts to almost pure guesses.

The shape of the post gives us a basic structure from which we can proceed. In essence we create two boxes to the left and right of the post (figure) and look for matching colors within those boxes. The simplest solution, however, is to build a blob of the matching color that contains the

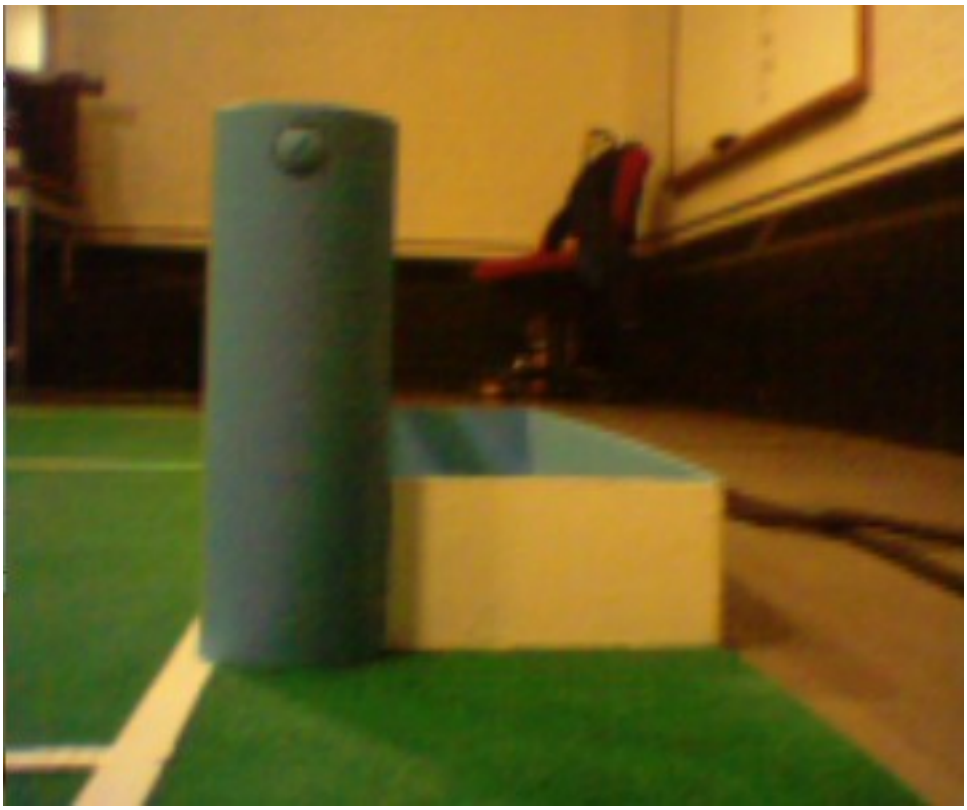


Figure 2: **Sometimes the backstop shows up on the wrong** side of the post.

post. In some cases this will clearly differentiate the post (e.g. the post is on one side of a large blob). Sometimes this will still not be sufficient. In such cases the things we check in order are:

1. Do we have a run of color on one side big enough to be another potential post?
2. Can we scan out significantly farther on one side than the other?
3. Do we have a preponderance of color on one side or the other?

If none of these things work it calls into question whether or not we are actually looking at a goal. Our field object data structure contains two certainty slots. One for how certain we are about the identification of the post, another for how certain we are that we actually have the object. In cases where we cannot determine which post we are looking at, at best we mark the identification slot as uncertain. In addition, if the post is relatively small we will actually throw it away. In practice this drastically reduced the number of false positive goals that we perceived.

2.4.3 Finding secondary posts

Once we are reasonably sure about which post we have it drastically reduces the possible locations of the second post. At this stage we throw out any runs of color which are not feasible from this point of view and proceed exactly as we did when finding the first post. Of course sometimes there will not be a second post. And sometimes the second "post" will actually be a backstop (figure). To screen for this eventuality we not only do the same filters that we do on the first post, but we also look at the relative size ratios of the two posts. This is not perfect as the ratios can be quite different depending on orientation (Fig. 3).

2.4.4 Finding backstops

While we use goalposts both for localization and for shooting, we use backstops exclusively for shooting. This changes the nature of our goals for backstop identification. First, backstop identification is only truly important when we are in position to shoot – among other things this implies that we have trapped the ball, which in turn implies certain things about our pose. Second, as opposed to the posts where we do not want to shoot at a blob of color, with the backstops our goal is to find the biggest blob of color and use that to direct our shot. The only tricky part is that we need to differentiate the backstop from the crossbar, and also identify when the backstop can be seen but is "blocked" (Fig. 4). In these cases we report that there is no backstop because the presence of a backstop implies that there is an available shot.

In the case where we have two posts finding the backstop is a simple matter of grabbing the largest blob that is between the posts but is not high enough to be the crossbar. In the case where we have only one post we essentially do the same thing but instead of using the second post that we found we project the location of a second post based upon where the first one is.

Once we have a potential backstop we need to find out what part of it is "shootable." To do this we do a series of vertical scans from the bottom of the screen up to the backstop. To be shootable there must be a "clear" area in the center of the visual field. For an area to be clear there must not be anything between the bottom of the visual field and the bottom of the backstop. Our algorithm starts by scanning up in the center of the visual field. If the scan can reach the backstop without intersecting an obstacle we continue the process in each of the left and right directions, continuing



Figure 3: **Two posts of very different size**



Figure 4: **There is no shot available**

in each direction until we either reach the end of the backstop or an obstacle is detected. This gives us two boundaries. We use the boundaries to determine whether or not a shot is feasible, and if not which direction we should move to get a better shot (Fig. 5). The only difficulty in doing the scans is differentiating between white patches that are dogs and white patches that are field lines. When we hit a white area in our scan we check against our line data structure for the presence of a line at that location. If we do not find one then we assume that there is an obstacle.



Figure 5: **A robot with this view should dodge left to get a better shot.**

2.5 Beacons

Beacons are the first field objects that we actively look for. Our search is based upon having found appropriate color transitions during our scanning phase. If a color transition of the right sort is found we use the same procedure that we use to grab colored rectangles while looking for posts. Essentially we grab two colored rectangles and check their relative sizes and locations.

2.6 Field Line/Corner Overhaul

The field lines this year changed from 2.5 to 5 cm in widthGiven that the field is larger this year, the increased line width was beneficial to our ability to detect lines accurately and consistently. While our team does not use lines to localize from, we do use corner information.

We rewrote our field line and corner detection and identification system in order to make the code more maintainable and general. For instance, our method that assigned identifications (e.g. yellow goal left L corner) in the old system contained 3252 lines of branching `if` statements; the resulting spaghetti code not only contained bugs but was all but unmaintainable. We decided that since we were writing our code in an object-oriented language (C++) we should make use of the benefits that provides. In particular, we now have classes to represent a line as it appears on the screen (*VisualLine*) as well as one that represents a physical line that exists on the field with a

set of fixed coordinates (*ConcreteLine*). We have a similar structure for our corners. Using these structures as well as more encapsulation and specialization of methods allowed us drastically reduce the amount of code in the file dealing with lines and corners, as well as increase the readability. For instance, the new code for identifying corners is 76 lines long. We reduced our the FieldLines file from 7976 lines of code to 4259, with a few hundred lines of code sprinkled throughout utility methods and the new classes.

2.6.1 High level overview

1. At certain columns and rows in the image, search for points that lie in the middle of white tape on green field
2. Using all the vertically and horizontally found line points, use a greedy algorithm to fit the points to lines. Lines must have a minimum number of points fit to them to be considered legitimate
3. Join line segments together into a single longer line if they logically fit together (have a very similar angle on the screen and pass other sanity checks)
4. Go through the list of points which were not fit to any line and attempt to fit them to the lines we have identified on the screen
5. For each pair of lines, calculate the intersection point and determine if it is a legitimate corner on the field (i.e. is not surrounded by green, the lines have a certain minimum angle between them)
6. If there is at least one corner and the robot is not goalie, do some checks to determine if the number of line segments or corners on the screen indicates that the robot is actually at the center circle and that the sanity checks in the creation of corners failed. If it is determined that we are at the center circle, throw out all the corners
7. If there are any corners left, attempt to identify which corner on the field the corner on the screen corresponds with

2.6.2 Line point detection

We use a simple Y-channel (brightness) edge detector to determine where green changes to white and vice-versa. In contrast to last year, we now make a distinction in the direction of change of the Y-channel. In particular, a green to white transition is marked by a sharp increase in Y-channel, which we denote as an *uphill edge*. A white to green transition is marked by a sharp decrease in Y channel, which we denote as a *downhill edge*.

Once we detect an uphill edge, we begin to keep track of the thresholded values we encounter until we reach a top/right edge. If we see too many undefined or pixels that are not white, we throw out the uphill edge we have seen and begin scanning for another uphill edge. Once we have both an uphill and downhill edge, we calculate the midpoint between them (either in the y or x direction depending on the scan) and add it to our list of line points.

One wrinkle in this is that there is not always a single uphill or downhill transition for each line; it is often the case that there is a two or more pixel transition between the green and white,

each pixel of which counts as an edge. For uphill edges we do not care; the first one we encounter is the one we wish to use because we want to accurately reflect the width of the line. For downhill edges, we cannot just accept the first downhill edge we find because that would often underestimate the width of the line. As such we wait a few pixels after finding a downhill edge before actually calculating where the midpoint is; this allows us to replace one downhill edge with another that comes right after it. These changes allow us to more accurately center the point within the line.

2.6.3 Create lines from points

While we find line points in two scans, one vertical and one horizontal, we do not do separate line creation passes. Instead we sort the horizontal line points (by x value, and then by y in the case of ties) and merge them with the already sorted vertical line points. We then use the one combined list in the creation of lines.

The basic algorithm for create lines is as follows:

1. Pick the first point in the list; call it the start to a line and add it to a separate list.
2. Go through the rest of the points, adding them to the list started from the first point if they pass a barrage of sanity checks; at a certain point (when the x-offset between the last point added to the line and the point being considered is too large) we stop considering any other line points in the list and consider what we have in the line so far.
3. Consider the size of the points list in the current line being created
   (a) If there are enough points in the list (3 currently), we create a *VisualLine* out of the list of points and delete all of these points from the list of all points we are considering
   (b) Else we say that no legitimate line starts with the line point we picked; advance to the next point in the list and start from step 2

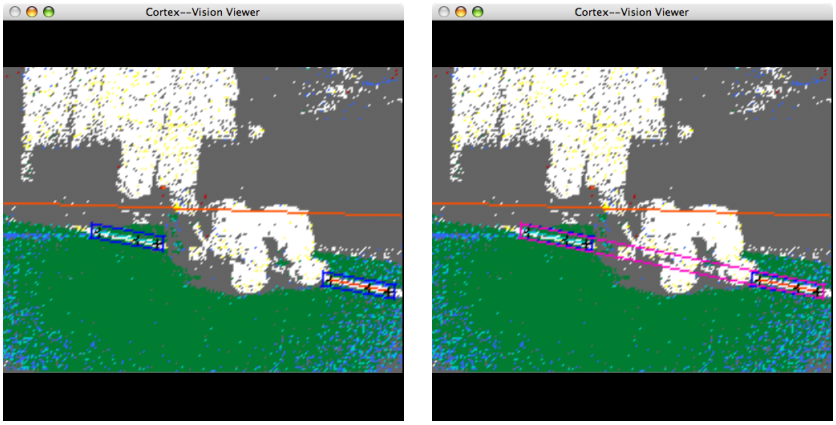
When creating a *VisualLine*, we run a linear regression on our set of points to obtain the equation of the line in question. We find this gives a better fit than simply using the left and right (or top and bottom) endpoints of the line.

2.6.4 Join line segments

We sometimes have one set of line points on the left of the screen, an obstruction to the line in the middle of the screen, and the rest of the points to the left of the screen. You can imagine a similar circumstance with lines that run top to bottom on the screen. See figure 2.6.4. Due to the x-offset sanity check in the `createLines` algorithm, the right most point on the left of the screen will not get joined to the left most point on the right of the screen.

We compare each pair of line segments and determine if they logically lie on the same line; if so we create a single line out of the line points of each constituent line, and delete the smaller line segments.

The process of joining lines allows us to be more stringent when originally making our lines, being confident in this method's ability to merge short line segments together.



(a) Before joining line segments (b) After joining line segments

Figure 6: An illustration of joining line fragments

2.6.5 Fit unused points

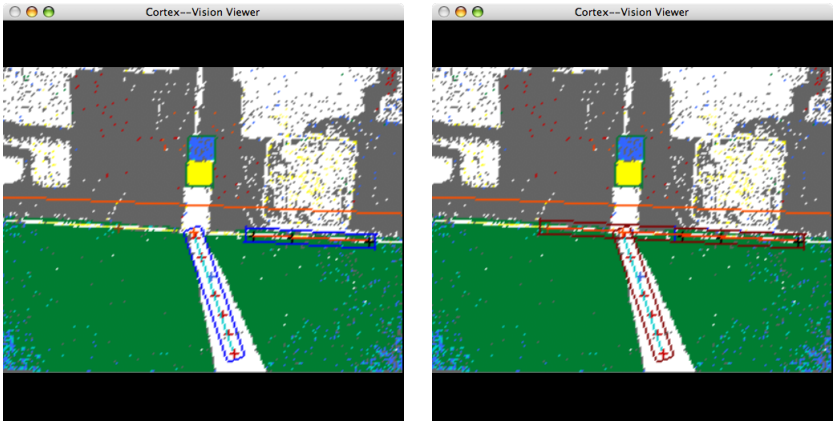
Due to the greedy nature of our line creation algorithm, we are not guaranteed to fit every point to a line, even if it legitimately belongs. For instance, consider the case of two near parallel lines, often found at the goal box. We might have two line points that line up vertically. Due to the sort employed (y value breaks ties), these two points will be considered sequentially. The first point will attempt to use the second point in the column as the basis of a line; for many reasons this will fail. At this point we say that, since there was no third point that fit, we will just eliminate the first point from contention. This allows us fail early in cases where we have an aberrant line point, but it also makes us miss some legitimate line points. That is where this method comes in.

For each point that we have left that has not been fit to a line, we compare it to the lines that we have identified. We determine how far off the line (in terms of pixels) the line point would be. If is beneath a certain threshold, we add it to the line and rerun the linear regression to obtain a new equation for our line. This has a good effect in practice; see figure 2.6.5.

Note that the unused point fitting could just as easily come before joining line segments. The reason we chose to do it afterwards is because we assume that after joining line segments, the longer lines will have more accurate regressions, and thus legitimate unused points will be more likely to fit the equation of the line well.

2.6.6 Extend lines

Due to our stringent checks for green on both sides of a line in the line point detection process, there are certain situations in which line points are missed consistently. For instance, a goalie panning to the side often has a goal post blocking part of the line, meaning the transition becomes green-white-yellow or green-white-blue rather than green-white-green. See figure 2.6.6 This can lead to corners being mistaken for L shaped rather than T shaped. For this reason we make the goalie do an extra method that attempts to extend lines as far as they can go. The idea is that we trace along the equation of the lines and if there is green on one side and white where the line hits, we allow this to be a new line point. We then fit these points to the line. This makes a drastic difference in the corner identification of the goalie. This method can be dangerous as it circumvents

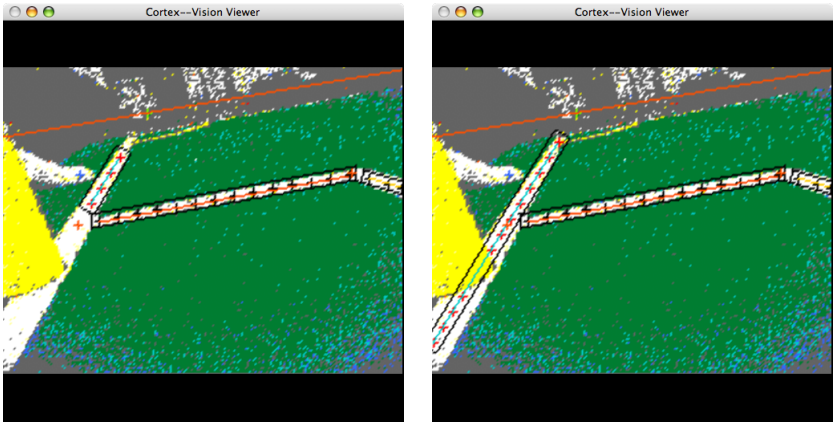


(a) Before fitting unused points, brown point on left is not a part of any line

(b) After fitting unused points, brown point has been included in horizontal line

Figure 7: An illustration of fitting unused line points

some of the sanity checks we have place in the line creation method; as such we allow it only to extend lines that are somewhat vertical in the image, and only for the goalie.



(a) Before extending lines, line points not detected in corner due to yellow post

(b) After extending lines, line points not detected in corner due to yellow post

Figure 8: An illustration of extending lines

2.6.7 Intersect Lines

We take our list of lines and pairwise compare them in order to calculate intersection points. If the intersection point is on the screen and not surrounded by green pixels (since actual intersections should be in the middle of white lines), and if the angle between the lines is approximately 90 degrees, we say that the intersection is a corner. We differentiate between inner L, outer L, and T shaped corners in our system. The inner/outer refers to the position from which we are viewing

the angle. For instance, a goalie in the goal box would perceive the L corners at the box as inner, whereas an attacker would see them as outer corners. These distinctions help us identify which corners we could be perceiving.

2.6.8 Center circle checks

Unfortunately, we do not make use of the center circle as a landmark for localization. Furthermore, due to the AIBO's point of view, line segments in the circle can appear to intersect with the long center line, causing corners to be formed. We detect when this happens with two sets of sanity checks, one that runs before object recognition, and one after. When we detect a situation that indicates we are at the center circle, we throw out all corners we have in the scene; it is too risky to allow them.

2.6.9 Identify corners

This is where the majority of the work was done in overhauling the system. The old system tried to enumerate special case logic and use positions in the screen and the angles lines made in order to determine which corner we were at. We decided we needed a more general, readable system. The basic algorithm is as follows:

For each corner identified in the image (`VisualCorner`):
    Get a list of all possible corners on the field (`ConcreteCorners`) for a given type
    (e.g. if the corner is a T, then there are 6 possible corners)
    For each `ConcreteCorner` that the corner could possibly be:
        For each visible object (beacon or goal post):
            Estimate the distance between the corner and the visible object
            If the estimated distance is too far from what the actual distance would be:
                Throw out the `ConcreteCorner` currently being evaluated

At the end of the method, we have each `VisualCorner` assigned a corresponding list of possible `ConcreteCorners`. When the list is of size one, we are fairly sure we know which corner we are at. Otherwise it is up to the localization system to figure out which of the possible corners is most likely. When we have a situation where all possibilities have been eliminated, we add all of the possibilities back into the list and pass off the completely uncertain corner into the localization system.

The main challenge faced was how to estimate the distance between a corner and a visible object. We use our pose information to estimate the location of the corner relative to the robot on the xy plane of the field. We do a similar calculation for the bottom-most point of the object in question. We then take the Euclidean distance between these two points as a rough estimate. This method is highly dependent upon the reliability of pose and the resulting matrix calculations, but ultimately worked well enough.

2.7 Transition from AIBO to Nao

We've made many adjustments to the AIBO vision code to run on the Nao. Most of them center around simplifying the algorithm, since there are fewer field objects, and the Nao camera is much

higher off the ground, and also has a much higher resolution. We've stripped out all the extra beacon and backstop code. Also, since the goals are much larger than on the AIBO, and the FOV on the Nao is smaller, we've made a greater effort to ensure we get smart distance estimates to goals which are obstructed by other robots or the edges of the frame.

2.7.1 Robot Detection

Since collision avoidance is going to be a crucial part of competing on the Nao, we've also started to work on opponent recognition. Since the resolution of the Naos camera is much higher, and its color quality is theoretically better, it is possible to detect other robots. If we are successful in this endeavor, the Nao league could open up a host of strategic possibilities with regard to opponents.

2.7.2 Camera Hack

It is worth mentioning that the prototype of the Nao that was available before RoboCup 2008 had a very unreliable camera. It suffered from two main problems. The first being that the camera often inverted the U and V channels in the YUV image. Our fix, which strongly resembled that of many other teams, was to check for the existence of the ball in the frame as the code started. If a ball was found, the channels were in the correct order. Otherwise, the channels were inverted, so we switched them in the code for the duration of the half. (This works of course only if you ensure the robot is actually looking at a ball when NaoQi loads your module).

The second problem was that the camera often gave no image at all (completely black). We never discovered a definitive fix for this issue, but either restarting the robot, or restarting the I2CSerial connector sometimes fixed the issue.

While we do not anticipate these issues with the next revision of the Nao robot, this information may still prove beneficial in the future.

2.7.3 Pose-based distance estimation and horizon line

Also changed from our AIBO codebase is the calculation of the horizon and the estimation of distances to pixels in the image using Kinematics (See Section 3.1). Although the overall approach remains the same (as is discussed in [3]), we have integrated this with an mDH parameterization of the Robot's kinematics.

To find the distance to a pixel, we define four coordinate frames:

- **Camera Frame:** Origin at the focal point of the camera, aligned with the head.
- **Body Frame:** Origin at the center of mass, aligned with the torso.
- **World Frame:** Origin at the center of mass, aligned with the ground.
- **Horizon Frame:** Origin at the focal point, aligned with the ground.

Each vision frame, we calculate the 4x4 (homogeneous coordinates) rotation matrices which translate between each of these coordinate frames from the mDH parameters and joint angles. When the vision algorithm needs the distance to a pixel, we project a line through the focal point and the pixel's location on the lens of the camera. If the line intersects the ground plane, we can find that point in the camera frame, and then translate it through the horizon frame into the world frame, and find the distance between the object and the CoM.

To find the horizon we project the horizon as the xy-plane of the horizon frame, and find the points of vertical intersection with the pixel grid on the lens. These points are then translated into the camera frame, and into pixel values. The pixel slope and location of this line are crucial to informing sanity checks and scanning direction for the vision algorithm. (See [3] for details).

3 Motion

On the AIBO, we did little motion work, since we employed a motion engine from UPenn 2005. Most of the work was concerned with building new scripted motions, like head scans and kicks to complement our behaviors. We also performed some work in optimizing the UPenn system and learning new gaits. For the Nao, we worked on our own walk engine, but ultimately decided to use the Aldebaran provided walk engine. Using their configuration options, we developed a stable walk that remained upright for all of the competition.

What follows is a brief discussion of the open-loop non-stability based (i.e. hand-made) walk which we developed in the simulator but ultimately abandoned for the Aldebaran walk.

The hand-tuned walk was an extension of the AIBO concept of using loci for each leg to define a walk. On the AIBO, the legs followed something like an ellipse shape in 3-space. When the legs were staggered correctly, and by modifying the shape of the ellipse, it was possible to generate omnidirectional gaits. The hand created walk we built did essentially the same thing. We defined a sequence of paths for the legs to follow, such that the legs alternated single and double support phases, and ultimately resulted in a "fast", omnidirectional walk in the Webots simulator (See video here `http://robocup.bowdoin.edu/media/2008/nao-walking.mpeg`). However, on the Nao, staying balanced is much more critical than on the AIBO, since there are only two legs, so we never tried the walk on the real robot. (We used the Aldebaran ZMP-based open-loop walk.) While we are currently working on a more robust, dynamically balanced walk, we will at least describe here the method of Inverse Kinematics we used in the hand tuned walk, and are also using for the dynamically balanced walk.

3.1 Forward Kinematics

Forward kinematics is conceptually and computationally much easier than inverse kinematics. Given a sequence of joint angles, it is always possible to give the resultant location of the end effector. The process could be easily done by hand in two dimensions, with pen, paper and a protractor. Given each angle and the length of each link, one could easily find the location of the end of the arm, for example. Inverse kinematics is the opposite of this process and can not be solved symbolically. (These parameters are specified in the Aldebaran RoboCup Edition Documentation)

Though there are many possible ways to calculate forward kinematics, we will use the modified Denavit-Hartenberg convention (referred to as mDH) , which is described online in many places (e.g. Chapter 1 of Springer Handbook of Robotics [5].) The basic idea behind mDH is to describe, in a standard way, the method by which joints are connected to each other. If all the links in the system are described in such a way, we can then write a general method to translate the joints for a given chain into a point in 3-space if we are also given the mDH parameters for that chain.

Generally, the mDH convention provides a standard way of converting between the local reference frames of each actuator to the next one in the chain. In addition, a usually small number of pre- and post-transforms define the offsets between the origin coordinate's frame and the first

actuator, and between the last actuator and the coordinate frame of the end effector, respectively.

In effect, each step in this conversion chain involves iteratively applying a rotation matrix to the origin coordinate frame, where the iterating matrix depends on the actuator's angle of rotation.

Using a symbolic mathematics package, such as Mathematica, we can pre-calculate the translation between the joint space and 3-space for each chain to save computation at run time.

3.2 Inverse Kinematics

Inverse kinematics is the opposite of forward kinematics. The process translates a point in 3-space into the requisite joint angles required to get the end effector to the appropriate location in 3-space. It is important to note that most points in 3-space cannot be achieved with a unique set of joints, and even more points are not reachable at all. For example, it is impossible to reach a point with your ankle that is further from your pelvis than the length of your leg. Similarly, there are many possible arm configurations which will ring a doorbell.

Solving inverse kinematics is basically a minimization problem in the error space of forward kinematics. Consider a target, T, in 3-space for the end effector: $T = (x, y, z)$. Consider an initial set of joints j_0, then we can apply forward kinematics (FK) to find the current location in 3-space, t_0, given those joint angles: $t_0 = FK(j_0)$. We can also measure the current error as the distance from the current end effector to the desired target $e(t_i) = (T - t_i)^2$. The initial error is given as $e(t_0)$. Now, we'd like to find a set of joints $j_2, \cdots, j_n$ such that each joint combination j_i has a lower error $e(t_i)$, until $e(t_n) \approx 0$. In this situation, the end effector for the joint angles j_n will have arrived at the specified location.

In practice, we solve this problem using Jacobians to find the "direction" of steepest descent in order to minimize the error as quickly as possible. We can perform this process quickly enough so that it can be used without trouble at run-time on the robot. An outline of the Jacobian approach is presented here [1]. For balanced walking, we also impose a second condition that the foot remain parallel to the ground.

Using inverse kinematics (sometimes referred to as IK), we can control all chains of the robot in 3-space, rather than the joint space.

3.3 Aldebaran Motion Parameters

For completeness, the code for our Python behavior module to set the Aldebaran walk parameters we used at RoboCup are given here:

```
#                              walkConfig                walkExtraConfig
WALK_STRAIGHT_CONFIGS = ((.04,.015,.04,.1,.018,.025),(5.85,-5.85,0.19,5.0))
WALK_TURN_BIG_CONFIGS = ((.04,.015,.04,.4,.018,.025),(5.85,-5.85,0.19,5.0))
WALK_TURN_SMALL_CONFIGS = ((.04,.015,.04,.1,.018,.025),(5.85,-5.85,0.19,5.0))
WALK_SIDEWAYS_CONFIGS = ((.02,.015,.02,.3,.018,0.015),(4.2,-4.2,0.19,5.0))
```

We did notice differences between different robots, which in some cases required changing the motion parameters depending on the robot.

4 Behaviors

We made large changes to both our individual and coordinated behaviors this year. These changes were brought about in order to improve our code organization and structure in the transition from the AIBO platform to the Nao. We found that our AIBO behavior code had become bloated and redundant after many revisions and additions, partly due to our loosely defined behavior structure. The code was cluttered with mixtures of functions which were undefined, undocumented, and out of place. This system created drag during development, and made for repetitive source code and difficult to maintain behaviors. Our new systems are extensible and facilitate faster and simpler development than the old structure.

4.1 Finite State Automaton

In behaviors, our largest change came in our building an abstract framework to handle the standard operations of the Finite State Automaton (FSA) structure used so often within our behaviors. This new FSA system keeps our code modular and organized. Building an API which clearly states how and when states within the FSA are exited and entered makes for simply defined state-based behaviors, with clear links between each state. Our new behaviors are written in Python, like our AIBO behaviors, chosen for its syntactical ease and its interpreted nature. As an interpreted language, Python does not need to be recompiled after changes. This allows us to reload modules individually without recompiling the underlying C++ structure. We first wrote an abstract FSA class to give our system its extensibility. This abstract class outlines the basic structure of an FSA and handles its execution. It holds the *run()* and *addStates()* functions, along with the state switching functions and other FSA helper functions. The *run()* function is called by our Brain behavior control module, at equal intervals, and has the responsibility of executing the current state. Along with these functions the FSA class maintains a timer and frame counter for the current state, a reference to the last state run, and a reference to the last different state run.

Each behavior is defined in two files, a FSA file that extends the abstract class and a state file. The FSA player file is responsible for initializing the FSA, setting implementation specific information, and storing helper methods. It calls the FSA *addStates()* function to create a listing of all the possible states for that behavior. *addStates()* uses Python's built-in *dir()* function to put all the states into a states array for later access. The states file contains all the states, with each state as a function. Here is an example of a basic state:

```
def spinLeft(player):
    if player.firstFrame():
        turn = motion.WalkTurn(150.,30)
        player.brain.motion.setNextWalkCommand(turn)

    elif (player.shouldWalkForward()):
        return player.goLater('walkForward')

    return player.stay()
```

There are three ways of switching between states: *goNow()*, *goLater()*, and *stay()*. *goNow()*

switches and runs the new state immediately, while *goLater()* switches states upon the next behavior cycle. As expected *stay()* will reevaluate the current state at the next frame. There is another function, *switchTo(newState)*, which is used to switch states from outside the FSA. The functions that define state switches, like *shouldWalkForward()*, are found in the FSA player file.

One main advantage of the FSA is its extensibility. It is very simple to add another FSA to the Brain module. The Brain only needs to construct the FSA and call its *run()* method once a behavior cycle. For competition in Suzhou, we chose to use three concurrent FSAs. This allowed us to break down larger behavioral problems into the smaller units. Our three behavioral components were a general player behavior, a head tracking controller, and a navigation controller.

4.1.1 Player

The general player behavior is the highest level behavior. It exerts control over the other two FSAs using *switchTo(***newState***)*. In competition, our player behaviors were very simple, but the player states are designed to execute the roles and subroles described in the next section. The player has states corresponding to its roles. For example, under the CHASER role, there would be states such as **chase**, **approachBall**, **confirmBall**, and **kick**. Each of these states uses the other two FSAs to achieve its goals.

Our behaviors for the 2008 RoboCup competition were developed very rapidly, almost entirely in the first few days in Suzhou. For competition, we had both our robots always in the "chaser" role with no communication between them. Our behavior was arranged by the Game Controller states.

1. **gameInitial:** The Nao stayed in a seated position.
2. **gameReady:** The Nao stood up.
3. **gameSet:** The Nao would pan its head until it found the ball.
4. **gamePlaying:** This was the main playing state, which was split into three categories of states.
    - **Find Ball:** The Nao would spin and look for the ball. If he saw the ball, he would stop and switch into an appropriate ball finding mode.
    - **Approach Ball:** The Nao will move so as to position himself in the appropriate place with respect to the ball and the opposite goal. Without localization, we needed a number of different states to achieve this positioning.
    - **Walk through Ball:** The Nao simply walks forward through the ball, in hopes of kicking it towards the opposite goal. After a preset amount of time, it switches back to finding the ball.
5. **gameFinished:** The Nao would stop moving and slowly sit into a safe position.

4.1.2 Tracking

The head tracking FSA controls all movements of the head, such as panning and object tracking. It holds states such as **tracking** and **scan**. The **tracking** state can track any Vision object, such

as a post, beacon, opponent, or, most importantly, the ball. The encapsulation of this functionality assists greatly in generating behaviors not only for game play but also from testing and training algorithms. While the ability to switch between the tracking of the ball and searching for landmarks from which to localize is extremely useful during games, the tracking of posts can be used to measure distance traveled, which will assist in such task as optimizing walking parameters or testing odometry or vision based distance estimation accuracy.

4.1.3 Navigation

Using the Aldebaran open-loop single-direction walk, we needed a system to control starting and stopping. This FSA takes care of the calculations and necessary repeated walk calls. Its states are separated into two categories, finite and infinite. States like **walkStraightForever** call on the motion engine to walk and then check to see if it has stopped before making another call. Other states take finite distances or angles and make the necessary calls to reach them. The navigator is designed to move the robot to any position (x,y,h) on the field. This allows for the higher level behavior FSAs to concern themselves only with when and where the robot should position, never how.

4.2 Coordinated Behaviors

We describe here the system of coordinated behaviors developed for use with the AIBO. Although the role-switching system was not used with the Naos during competition, the system was ported to the Nao platform and will be used for RoboCup 2009 after parameters are tuned for use with the individual behaviors developed prior to competition.

The intention of our behavioral system was always in facilitating a robust coordinated system to allow for high level team play. We developed a decentralized, dynamic role switching system, which utilizes the concepts of **strategies**, **formations**, **roles**, and **sub-roles** to build a robust system of position, based soccer.

4.2.1 Strategies

Strategies are the first and thus most abstract level of our coordinated behavioral system. Strategies specify a general manner of play to be used. For competition last year we had only one strategy **sWin()**. However, we have since then created more strategies and are hoping to have many in place for RoboCup 2008. A strategy is theoretically a set of formations, which all share some underlying trait. We could define a strategy **sDefensive** which would require that every formation within it had two defenders. The logical extensions of how to build additional strategies are fairly obvious; however, the decision making process for determining which strategies should be used is a much more complex problem and is a target of current research within our team.

4.2.2 Formations

For competition we had the following formations: *Normal*, *Kickoff Play*, *Kickoff*, *Goalbox*, and *Finder*.

Formations act as a layer above the role switching system to allow for different roles to be specified for selection by the agents as well as sometimes dictating specific roles when well known situations arise. The formation *Normal* is used for regular game play and has the most advanced

role switching as described in section 4.2.3. Although the agents use *Normal* for the majority of the time the first formation called will always be *Kickoff Play*.

The kickoff in any soccer game is the situations where the information will be most clearly known to all members of the team: position and velocity of the ball, relative position of the opponents (depending on which team is kicking off and which is defending), and near exact positions of all teammates. Thus prior to kickoff robot A will always setup in position to start playing defense, robot$B$ will be in the center ready to become the chaser, and robot C will be on the wing ready to become the attacker. Once play begins all robots will maintain their roles for a specified amount of time, so that robot A will move immediately to his defensive position, robot B will chase the ball, and robot C will move up the wing ready to receive a pass shot forward by robot B. The robots will move out of this formation and into *Normal* after a short amount of time or at any point, when something unexpected occurs (i.e. the ball goes out of bounds, a player is penalized, etc.), but it should be pointed out, that play in the *Kickoff Play* does not look any different then other points in the game, the formation simply reinforces the method of soccer we wish to be played against any poor data which could occur during the very familiar kickoff situation. The *Kickoff* formation exists as a fall back in case one or more robots are not on the field at the beginning of play and specifies different positioning setups for having only one or two field players.

The *Goalbox* formation is put into action whenever the ball moves into the teams own goalbox at which point the goalie becomes responsible for controlling the ball and all three field players stay out of the goalbox, so that an 'illegal defender' penalty is not incurred. In order to ensure that at least one robot will go for the ball, the goalie will chase the ball if the ball comes within the area of his goalbox extended five centimeters in all directions, while the field players will not attempt to chase unless the ball moves more than five centimeters into the box, giving an effective ten centimeter buffer. For further insurance the goalie will begin to chase if the ball is relatively near his goalbox and no other players are claiming to be chaser, effectively signaling that they believe the ball to be inside the goalbox. Once put into action the formation dictates, that two of the field players stand to the side of the goalbox in hopes of receiving an outlet pass from the keeper while the third robot waits near midfield ready to get any balls placed back in at midfield after going out the end line. The roles here are prescribed as *DEFENDER*, *DEFENDER*, and *ATTACKER* with the sub-roles for the two defenders specifying that one should maintain a left deep back position, while the other plays a right deep back role.

The *Finder* formation acts as a global fall back in case all robots lose track of the ball's position. If no robot has seen a ball within the past five seconds (150 frames or 30 teammate report cycles) all robots will move to predetermined positions on the field in order to optimize viewing of the entire field in hopes of finding the ball, two robots will be placed in static positions while a third robot begins to roam a predefined path. Like all other formations *Finder* is a non-negotiated decision, a robot begins searching if it has processed any ball information, either images with identified balls or reports of seen balls from other teammates.

4.2.3 Roles

Roles define the fundamental activity a robot should be preforming from frame to frame and directly influence which behavioral states in the player's FSM are to be used. The five roles used at RoboCup 2007 were *CHASER*, *ATTACKER*, *DEFENDER*, *SEARCHER*, and *GOALIE*. These roles are fairly broad in scope (except for searcher which is used only for the *FINDER* formation) and thus we define a number of sub-roles to further control the team dynamic.

4.2.4 Sub-Roles

For further specifying what function a robot should serve on the team, we define sub-roles beneath the standard roles. Sub-roles facilitate two separate necessities in our system. First they specify parameters which refine the objective of a robot at any given time; for the most part this takes the shape of dictating positions for players based upon the location of the ball. However, the notion of a sub-role is quite abstract and the implementation of one sub-role can be very different from another. For example there are two sub-roles for the chaser role *CHASE_NORMAL* which says the standard chase behavior should be used and *CHASE_AROUND_BOX* which is only used when the chaser's own goalbox lies between the chaser and the ball. When *CHASE_AROUND_BOX* that potential fields be used for finding a path that will keep the chaser from incurring a penalty for illegal defender.

The second task accomplished is that sub-roles allow multiple robots to have the same role without conflict between the two players. Thus we can have two defenders who do not fight to maintain the same position in front of the goal, but instead take positions which allow them to better utilize their knowledge that there is another defender assisting in the overall defensive objective.

4.2.5 Role Switching

The vast majority of play occurs inside of the *Normal* formation with that in mind the role switching system is based almost entirely around the idea of having the roles of *CHASER*, *DEFENDER*, and *ATTACKER*) to be allocated. The role-switching during *Normal* operates around our philosophy of having one, and only one, robot chase the ball at any given time, while allowing any robot, save the goalie, to become *CHASER*. The *CHASER* role is allocated to the robot, which is determined to have the minimum *chase time*. We define *chase time* as the estimated time it will take a robot to get to the soccer ball, calculated primarily around odometry. We assume that the robot is unobstructed and not delayed in moving to the ball for the calculation of *chase time*. To reflect our overall aims of scoring a goal, we reduce the assumed *chase time* in special circumstances – such as an agent lined up behind the ball going towards the opponent's goal. Lowering the *chase time* improves the likelihood that an agent not closest to the ball, but still in the best position to score, will become *CHASER*. Each robot broadcasts its *chase time*, when communicating to teammates, allowing each robot to determine the chaser independently and thus denying the need for any negotiated decision making. We feel that a non-negotiated role switching system is absolutely necessary to uphold our philosophy of reaching the ball as fast as possible, waiting just five frames to receive confirmation to chase could often cause a robot to loose the ball to opponent at high levels of play. Information lag and system noise incurred from sensor data may cause the agents' divergent world models to bring about different conclusions as to which robot has the shortest *chase time*. To combat this error while maintaining our non-negotiated system, we use a tiebreaker, which draws its inspiration from real sports, where certain players "call off" others (i.e. a goalie can call off a defender). For our system we use player numbers (each robot has a unique number 1-4), such that in a tie-break situation the higher player number can call off a lower player number.

If Robot A believes it might be the fastest to the ball (it is within a small threshold ϵ of the minimum *chase time* for all the robots), it will start pursuing and calling the ball ("I got it!"). Robot A will continue to chase the ball until a higher ranked robot calls it off, and/or any lower ranked robots receiving Robot A's call discontinue chasing. Thus Robot A is committed to

chasing the ball. It cannot stop chasing and calling the ball until its *chase time* is outside a larger threshold δ (that is $\delta > \epsilon$). This prevents hysteresis, where robots oscillate back and forth between roles due to small changes in the data. There is a third important threshold, λ, which ensures that a robot should stop listening to a higher ranked teammate if its *chase time* is less than that teammate's *chase time* by the value of λ. This ensures that when there is a discrepancy between local information and communicated information the robot relies on its local information. When things rapidly change on the field, a robot must not wait for a message from the current *CHASER* with higher rank before it acts. In summary Robot A will chase the ball if:

1. $chasetime(A) - min(chasetime(A, B)) < \epsilon$ and no higher robot is calling off A, or
2. $chasetime(A) - min(chasetime(A, B)) < \delta$ and it was already chasing and no higher robot is calling off A, or
3. $chasetime(A) < chasetime(B) - \lambda$ where the Robot B is the higher ranked robot calling off A

Each threshold controls a feature of the robots' cooperation. Increasing ϵ makes it more likely that a robot who is farther from the ball will ultimately end up being *CHASER*, but makes it less likely that no robot will be *CHASER*. δ controls how willing robots are to switch roles. Increasing the δ value decreases how often the robots will switch roles, which can leave the wrong robot chasing the ball, but protects against robots oscillating back and forth about who should chase. λ controls how much a robot should rely on local information. Increasing λ makes it less likely two robots will chase the ball, but slows down reactions to a ball suddenly being closer to a non-*CHASER* robot.

After deciding on which robot should become the *CHASER*, the remaining field players must decide which robot is to become the *DEFENDER* and which should become the supporting *ATTACKER*. The decision making process for this issue is the same as in determining the *CHASER*, only the determining metric is *distance to own goal* instead of *chase time*. We structure the tie-breaking thresholds about defense to ensure there is always a *DEFENDER* when communicated data deteriorates.

For all other formations the roles would be determined based on finding the least distance to be moved by all three field players to move into the three desired positions.

4.2.6 System Breakdown

When communication was disrupted, something which was prone to happen at competition, the role switching system could not properly function. To combat this problem we built a fail-safe behavior into the system which would take over when a robot did not receive any communication data from his teammates for a decent amount of time. The robot would play a defensive position, but still chase and attempt to clear balls which came into the back third of the field. To keep all robots from running into each other during a complete communications break down, the player number would be used to dictate a unique central position for the backup role.

4.3 Playbook Editor

The system of coordinated behaviors described above allows for countless numbers of strategies, formations, roles, and sub-roles, simultaneously viewing the various formations is very difficult and

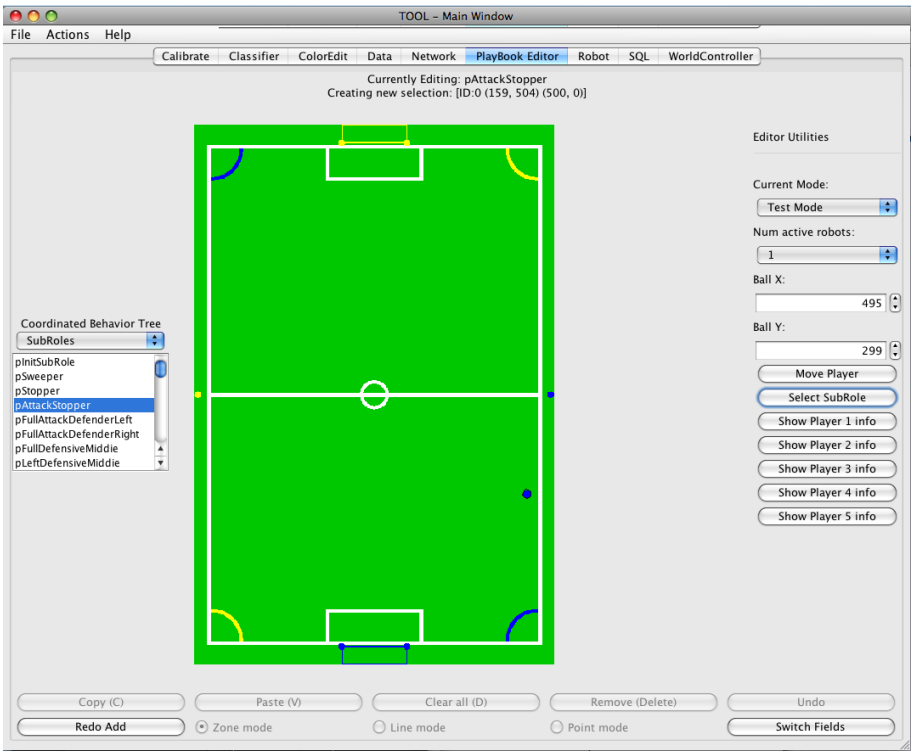


Figure 9: The Playbook Editor shown displaying a sub-role.

rather expensive, since a full team of robots must be used in order to see how the robots will react to any given situation. As such, we began work on a Playbook Editor last year, which addresses both issues.

The core functionality of the Playbook Editor is the ability to output specific components of the behavior tree via a GUI. This GUI takes the form of a field, where one can draw desired positions and generate Python code corresponding to these positions, which is immediately usable within the greater coordinated system. Since this was the most vital feature for use last year it was the only portion fully-functional at competition. However the remainder of the component generation system has significant work completed and mostly only lacks the ability to generate correctly formatted code. Below we continue with a description of the remaining parts of the system.

At the level above this position generation one can group how a specific role can use the various sub-roles based on ball position. Similarly a formation can be chosen as a set of roles and strategies can be built in a way very close to that of roles, where specific formations are selected for when the ball is within specific regions of the field. This functionality is scalable to take into account situations in which some players are not on the field.

The second portion of the Playbook Editor is a rudimentary simulator built to help show the positions of the robots given specific ball positions on the field. This allows one to choose a specific strategy or role and place the ball in various positions on the field to view how the robots will act. This functionality helps greatly reduce the amount of time and energy put into testing team positioning, running the entire test offline.

5 Current and Future Work

Three members of this year's team are working on RoboCup related senior honors theses. We are developing a closed-loop humanoid walk engine, to be used instead of the provided Aldebaran engine. Another member is replacing our Extended Kalman Filter-based localization with a particle filter which will also perform opponent tracking. The third project concerns itself with designing a unit testing system for objectively measuring the efficiency, robustness, and accuracy of our vision system.

5.1 Vision

With the overhaul of the field lines system and the transition to a new robot platform, there were many changes to the vision system. A large quantity of time was spent ensuring that recently checked in changes had not broken something, and if they had, tracking down what had broken, and where in the code it occurred. One student is developing a system that will plug into the offline debugging environment we have in order to allow users to specify what objects are visible in frames, and where they occur. The system will then create unit tests that can be automatically run whenever new code is added to the code base to ensure that nothing has broken. Furthermore, the system will serve as a testbed for machine learning different aspects of the vision system.

We also plan on continuing to refine our object recognition algorithms, especially those involving opponent recognition.

5.2 Motion System

Since we felt limited by the closed-source Aldebaran walk we used in the last RoboCup, we are developing are own dynamically balanced walk which will address more directly our needs in a soccer environment. The Aldebaran walk provides an interface to effectively queue a sequence of footsteps in a few chosen directions (including turning). However, it is highly ideal in the RoboCup environment to have a truly omnidirectional gait, which can be controlled with the three tuple speed vector (x, y, θ).

Our work builds off the ZMP-based preview-control method developed by Kajita and Czarnetzki (of the BreDoBrothers) [2],[4]. The main goals of the project are to develop a simple, dynamically balancing closed-loop walking system. The thesis will also focus on being a comprehensive guide for re-implementation so other teams from small undergraduate programs without control theory or explicit engineering courses will have an easier time getting started in the Nao SPL league.

5.3 Localization

Although the Extended Kalman Filter (EKF) based localization system of our 2007 code base was available for use on the Naos, our behaviors were to primitive to utilize it. This year we are moving away from the EKF to a Monte Carlo Localization system. We have decided upon this do to the particle filter's inherent ability to deal better with negative information and ambiguous landmark data, two issues we find to occur far more often with the Nao field, than they did within the Four-Legged-League. Along these lines we are also undergoing preliminary work in building a probabilistic model of opponent position and movement. We hope that accurately tracking opponent positions will induce a more soccer-like style of play, more in line with the stated goals of RoboCup than some of the play style developed on the AIBOs.

5.4 Behaviors

Behavioral work this year will build upon the new architecture built for the Nao last year. Direct aims of this year are to incorporate the world-modeling information more directly. The one main place this will occur is in implementing the role switching and coordinated behavior systems which were extensively developed for use with the AIBO. To help actualize this we plan on completing the Playbook Editor as described above. Beyond this we hope that having a wider number of scripted motions and improved stabilization will allow us to perform many more complicated tasks than last year.

6 Code Diff

Our Nao code base is entirely our own, so we do not believe a code diff is necessary.

7 Code Availability

The Northern Bites are committed to giving back to the RoboCup community. As such, we our allowing access to the stable branch of our code base, while under development. The code is available online at `http://github.com/northern-bites/`

References

[1] Samuel R. Buss. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. `http://math.ucsd.edu/~sbuss/ResearchWeb/ikmethods/iksurvey.pdf`, April 2004.

[2] Stefan Czarnetzki, Sören Kerner, and Oliver Urbann. Observer-based dynamic walking control for biped robots. *Pre-print Elsevier*, 2008.

[3] Kenny Hong. Nubots: Enhancements to vision processing, and debugging software fo robocup soccer. Thesis for the University of Newcastle, 2005.

[4] Shuuji Kajita, Fumio Kanehiro, Kenjio Kaneko, Kiyoshi Fujiwara, Kensuke Harada, Kazuhito Yokoi, , and Hirohisa Hirukawa. Biped wlaking pattern generation using preview control of zero-moment point. In *International Conference on Robotics and Automation; Proceedings of the 2003 IEEE*, 2003.

[5] Bruno Siciliano and Oussama Khatib. *Springer Handbook of Robotics*. Spring, 2008. `http://books.google.com/books?id=Xpgi5gSuBxsC`.