USING IMPROVED OBJECT DETECTION IN ROBOCUP SOCCER FOR COLLISION AVOIDANCE AND RECOVERY

NSF Summer Undergraduate Fellowship in Sensor Technologies
Bill Sacks (Computer Science) – Williams College
Advisor: Professor Jim Ostrowski

ABSTRACT

RoboCup legged soccer is an international soccer competition between autonomous, four-legged robots. In past competitions, collisions between robots were common. My objective, therefore, was to augment the robots' sensor systems to allow them to avoid and recover from such collisions; this would both speed up the game and enable the robots to dribble the ball around other players. I have increased the accuracy and stability of the ball's position estimation and have added a calculation of the ball's relative velocity. Using this velocity, I have developed an algorithm to determine if a robot has collided while trying to get to the ball; if the velocity remains near zero for too long, the robot will try another path to the ball. I have also greatly increased the accuracy of the detection of other robots and have added the ability to store the positions of all other players on the field. A dribbling robot will then use these position estimates for collision avoidance, using the method of artificial potential fields. I describe a number of modifications that were necessary for this method to work well. Finally, for both the ball and other players, I have incorporated readings from the robots' infrared distance sensors, and have added an update of the relative position of an object based on the observer's own movements.

1. INTRODUCTION

RoboCup is an international soccer competition between autonomous robots. The aim of this annual competition, now in its fifth year, is to provide a well-defined task - winning a game of soccer - that serves as a platform for research in many areas of artificial intelligence and robotics. Good performance in RoboCup requires the application of techniques from the fields of real-time planning, multi-robot coordination, computer vision, and real-time sensor fusion, among others [1]. The RoboCup environment has a number of advantages over real-world environments for research in these areas. First, since it is a controlled environment, many of the complexities and uncertainties of the real world can be eliminated. For example, the need for complex image processing is eliminated by color coding every object on the field, and the number of other robots on the field is known at all times. In addition, it is easy to measure the performance of a system - by how often it can win a game of soccer. Finally, the entertainment value of this task cannot be overlooked. RoboCup generates much more interest than other, more theoretical research tasks, and thus has the potential to create a greater public interest in AI and robotics research.

There are currently four RoboCup leagues; three of these involve actual robots, and one involves simulation. The University of Pennsylvania participates only in the legged robot league, and so this paper is limited to a discussion of that league. The robots in this league are Sony's AIBO robots; this year we are using the ERS-210 model. These are fully autonomous, four-legged, dog-like robots (Figure 1). Each leg has three degrees of freedom, as does the head. The robots have touch sensors on their head, chin, back and feet, as well as an acceleration sensor. Although the acceleration sensor is not very accurate [2], it is sufficient for determining whether the robot is upright or on its side. The vision system consists of a CMOS camera and an infrared distance sensor, which determines the distance to the object in the center of the camera view. Each robot also contains a speaker and two microphones, allowing simple communication between teammates [3]. We use a developmental version of the AIBO, which is programmed in C++.

In the legged league, the soccer game is played on a two-meter by three-meter field, with three players on each team; one player on each team is designated as the goalie. All processing and sensing is done on-board. The only communication between teammates is through their built-in speakers and microphones.

Partly because the robots are relatively large compared to the field, and partly because three or four robots are often headed toward the ball at once, collisions between robots are common. The referee can, in many circumstances, separate robots that have collided after a set time, but these frequent collisions nonetheless slow down the game considerably. A robot with the ability to avoid collisions, and to recover from any collisions that do occur, would have a clear advantage over one that cannot do so. It would be able not only to reach the ball more quickly and more reliably, but also to dribble the ball around the other robots on the field.

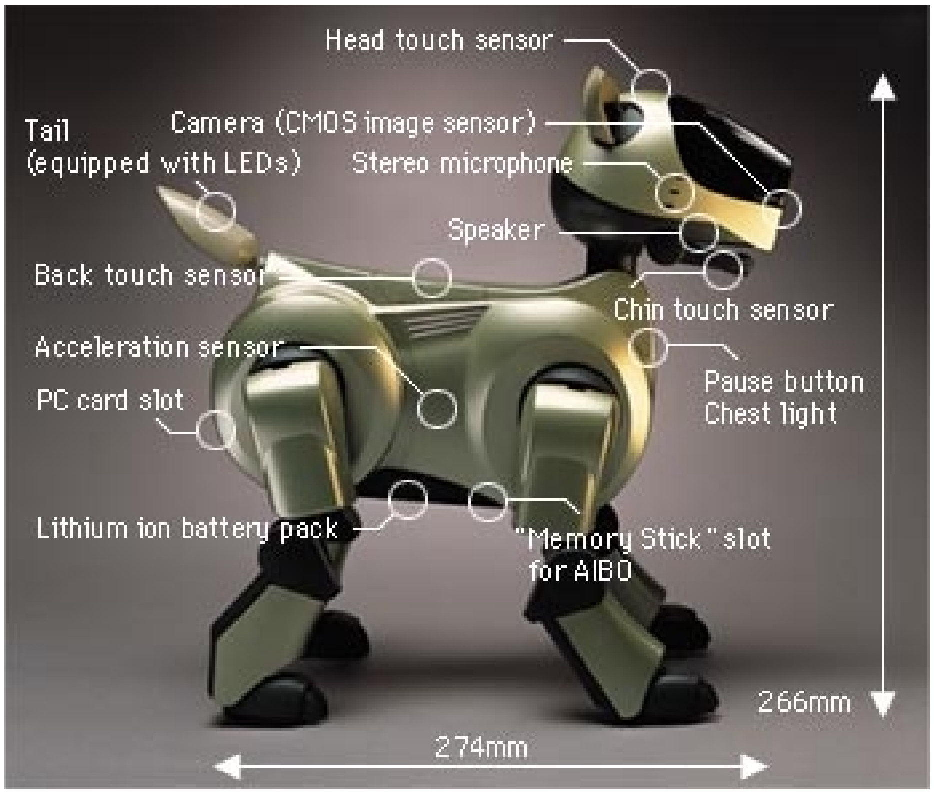

Figure 1: Sony's AIBO ERS-210 (from [3]).

With this in mind, my goal this summer was to extend the University of Pennsylvania's robots' vision and detection systems, and to use these advanced detection capabilities for collision avoidance and recovery. To achieve these goals, I have primarily used two methods: an estimation of the ball's velocity to determine whether a robot has collided while trying to get to the ball, and the technique of artificial potential fields to dribble around other players once a robot has control of the ball.

2. BACKGROUND - PENN'S ROBOCUP SOFTWARE STRUCTURE

Penn's software structure is unique among software for the RoboCup legged competition in its great modularity. The code is essentially composed of three pieces - the high-level planner, the Extended Logical Sensor Module (XLSM), and the Extended Logical Actuator Module (XLAM) (Figure 2). These three components communicate with each other and with the hardware through a number of shared data objects. For example, there is an object holding the robot's own position and other state variables, and another object representing the ball's position. Positions are recorded in polar coordinates, storing the distance and angle[1] from the observer. Most objects also carry associated confidences, which are set to 100 when the object is observed and degrade over time when the object is not observed.

[1] 0° denotes straight ahead.

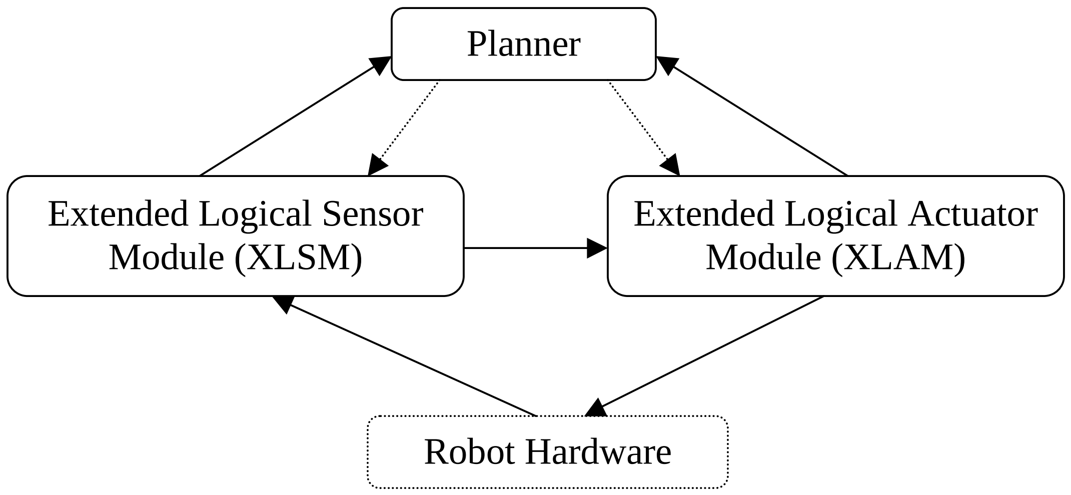
Planner
Extended Logical Sensor Module (XLSM)
Extended Logical Actuator Module (XLAM)
Robot Hardware

Figure 2: Top-level view of overall structure. Solid arrows indicate data flow; dotted arrows indicate commands (from [2]).

The XLSM is further broken down into different extended logical sensors (XLSs) for each class of objects on the field. There are, among others, XLSs for the ball, for teammates, for opponents, and for the robot's own state. Each XLS is responsible for generating the current best estimate of its corresponding object; other modules can then use this information without knowledge of the implementation of the XLS. Similarly, the XLAM is broken down into different extended logical actuators (XLAs) for the different pieces of robot hardware that can receive commands. Thus, there are XLAs for walking and for moving the head, among others. The planner can call upon an XLA to perform some action, such as walking to the ball, without needing to know exactly how that action will be carried out [2].

This modularity greatly simplified my task. I was able to focus primarily on the XLSs for the ball, the opponents (the "foe XLS"), and the teammates (the "pal XLS"), as well as the high-level planner, without having to worry about the details of, for example, exactly how the various walk commands would be executed. In addition, this modularity made it easy to divide the complex task of obstacle avoidance into smaller pieces. Because of the clear separation between the detection of obstacles and the higher-level planning needed to avoid these obstacles, these two components could be written and debugged independently.

The design of the various XLSs was further simplified by the lower-level code already in place. Especially useful was an algorithm to find the size and position of the four largest blobs of any color in a given camera image. In addition, there was code in place to filter out-of-range readings of the infrared distance sensor (under 10 cm or over 90 cm), and to allow a robot to adjust the estimated relative position of objects based on its own movements (although this capability was not fully utilized, as described below).

3. USING THE BALL'S VELOCITY FOR COLLISION DETECTION

In the games we observed robots frequently would collide when going after the ball, and then get stuck, with each robot trying to push past the other. As long as the ball was not moving, the players would continue trying to move in the same direction, and none would move anywhere. Since this usually occurred when the robots were tracking the ball - our robots already had the ability to move their head to keep the ball in the center of their vision - we decided to make use of this constant observation of the ball to help the robots determine when they were stuck. Specifically, we decided to use the ball's velocity as an indicator that a robot is stuck. If a robot is trying to move toward the ball, but the ball is not getting closer, then the robot is probably stuck.

3.1.1 Advances in the Ball's Detection System

Before I could determine the ball's velocity, I had to improve the ball's sensor system. Under the old system, the angle to the ball was determined using a combination of the robot's head position and the location of the ball in the camera image. In most cases, the distance to the ball was determined using a calibration table that mapped the number of pixels of the ball image to its distance. When the ball was almost directly under the robot's head, the distance was instead determined using the equation:

distance = (height of dog)/tan(tilt angle to ball)

This system for determining the ball's position worked fairly well, so I decided to keep it in place. I tried replacing the calibration of distance based on size with readings from the infrared distance sensor, but found that the resulting distance estimations were actually less accurate than the size calibration when the ball was more than about 40 cm away.

In a few situations, however, the robots still obtained inaccurate ball readings using size calibration. One was when the ball was partially obstructed, because an obstructed ball would seem farther away. We were already guarding against this to a degree by making two calculations of distance, one based on the ball's height and the other on its width, and choosing the closer distance. This worked well most of the time, but would still produce inaccurate readings when the ball was obstructed in both dimensions. To remedy this, I decided to use readings from the infrared distance sensor whenever this sensor provides a reading that is less than 2/3 of the distance obtained from the size calibration, thus indicating that the ball is obstructed. Despite the fact that the infrared sensor is less accurate than simple size calibration, in this situation it is better not to use the ball's size to determine its distance. Although the ball has to be in the center of the camera image for this to work, this is often the case since our robots are usually in a mode in which they track the ball, keeping their vision centered on it.

We also had problems with the ball's distance suddenly jumping to much larger values. This usually seemed to occur when a robot lost sight of the ball and saw a small patch of a similar color, either outside the field or on another object on the field. I made two small changes to fix this problem. First, I added a check of the height of the supposed ball - if it is much above the horizontal, it cannot actually be the ball, and so should be ignored. In addition, rather than using a single threshold for the minimum pixel area of the ball, I created two thresholds. If a player has recently seen the ball nearby, it uses a larger area threshold than if it either has not seen the ball recently or last saw the ball relatively far away.

After these modifications, our robots were able to determine the ball's position more accurately. But one major flaw remained in this estimate, arising from our use of relative, rather than absolute, positions of objects on the field. When a player did not currently see the ball, and was moving relative to the ball, it made only very rudimentary adjustments to the ball's position. It only updated the direction to the ball, and this update was based solely on the robot's rotational movement, not its translational movement. I expanded these updates to modify the ball's distance in addition to its direction. I also included updates based on the robot's translational movement.[2] Our players can now maintain a relatively accurate estimate of the ball's relative position even when they are moving (assuming the ball itself is not also moving).

Despite these advances, the ball's position was still not entirely stable. This was due primarily to the inaccurate nature of the low resolution camera we use. A single pixel's difference in the size of the ball can cause its distance estimate to change by 5 or 10 cm in some cases. In order to get a relatively stable reading of the ball's velocity, as well as of the its current position, I added a second, filtered position estimate. This filtered data is obtained simply by averaging the last four estimates of the ball's position,[3] resulting in the average position over a period of about 1/4 second.

3.2 Calculation of the Ball's Velocity

With this filtered data in place, a relatively accurate calculation of the ball's velocity was, in theory, more feasible. To determine whether a robot is stuck, only the radial velocity is needed, but the angular velocity is useful for other purposes, such as helping the goalie decide how to block an incoming ball. I therefore decided to calculate both components of the ball's velocity.

Velocity is calculated by checking the ball's position[4] every 1/3 second. Checking more often than this would probably yield inaccurate results, as changes in the ball's observed position would be due more to sensor errors than to any real movement. If a robot saw the ball in the last 1/3 second, and had previously seen the ball within the last 1.5 seconds, it estimates the ball's current velocity by dividing the difference in positions by the time between the two readings. If it had not seen the ball for more than 1.5 second before, it

[2] A robot does *not* update the ball's relative position if it is currently dribbling the ball.
[3] Because of the difficulty of averaging multiple angles, the positions are converted into Cartesian coordinates, averaged, then converted back to polar coordinates.
[4] For the radial velocity I use the filtered data, but for the angular velocity I chose to simply use the unfiltered data. Angular estimates tend to be more stable over time than distance estimates, and using the unfiltered data gives a more current velocity estimate.

stores the current position but does not calculate the velocity, as the velocity over such a long time period is likely to be inaccurate.

3.3 Collision Detection

Once a robot can calculate the relative velocity of the ball, it can determine whether it is getting any closer to the ball. If the ball's relative velocity remains near zero - or if the ball is actually getting farther away - for a significant amount of time while a player is trying to move toward it, then that player is probably stuck. Using this reasoning, I created a "stuck" condition that is tested repeatedly whenever a robot is trying to approach the ball. Each time the condition evaluates to true - that is, each time the ball's radial velocity is near-zero or positive (moving away from the observer) - a counter is incremented. If the counter reaches a given threshold, this signals that the robot is probably stuck and should change its direction. The counter is decremented whenever the ball's radial velocity is significantly less than zero (getting closer to the observer).[5]

It quickly became apparent that, while this system worked well when a robot was far from the ball, the robots would often think they were stuck when they were about to reach the ball. This was because our robots take smaller steps when they get close to the ball to avoid inadvertently kicking it away, and thus the ball's relative velocity drops close to zero. To remedy this, I added a second counter tracking the amount of time the robot has been within 35 cm of the ball - the distance within which its step size begins to decrease. A robot within 35 cm of the ball will consider itself stuck only if this second counter reaches a given value.

This system for determining whether a robot is stuck has performed well in testing and in practice games. The velocity reading is still not very accurate, mostly due to uncertainty in the ball's distance estimate. Even a difference of 5 cm between successive readings (1/3 second apart) will yield a velocity of 15 cm/sec; for comparison, our robots' fastest speed is about 12 cm/second. When averaged over a number of readings, however, the velocity is accurate enough to determine whether a robot is stuck. Our robots can almost always determine that they are stuck within a few seconds of running into an obstacle. They still sometimes believe they are stuck when they are not, usually when the ball is moving away from them or when they are turning slowly toward the ball. This happens infrequently, though, and preventing it would require raising the counter thresholds. This, in turn, would mean longer delays between a robot's hitting an obstacle and its determining that a collision has occurred.

3.4 Collision Recovery

Once a robot determines that it is stuck, it must act to free itself. To do this, it first takes ten steps backwards. While this is more than is usually necessary for a player to disentangle itself, this large backwards movement helps ensure that the player will not immediately get stuck again.

[5] There is an added restriction that the counter can never be negative.

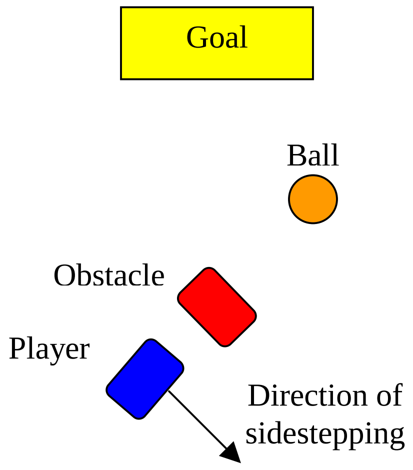

Figure 3: Direction of sidestepping after a collision.

Next the robot decides whether it should sidestep or simply rotate. In most cases it will sidestep to try to get around the obstacle. The direction in which it sidesteps is determined using the relative positions of the ball and the opponent's goal. The robot tries to keep the ball between itself and the goal (Figure 3). While this method does not always get the player around the obstacle, it works well enough for our needs, and has the added benefit of positioning the player to shoot when it reaches the ball.

If the angle to the ball is greater than 30°, the robot does *not* sidestep after backing up; instead, it rotates to face the ball. This often occurs when the robot is on the edge of the field, facing out, and is prevented by the wall from rotating to face the ball. Thus, the robot need only back away from the wall and face the ball.

After sidestepping or rotating, the robot will again try to walk to the ball. Although it sometimes takes a few tries to get around an obstacle, this method of collision recovery seems relatively effective. More work could be done, however, on developing a heuristic to determine the best direction to sidestep after backing up.

4. AVOIDING OTHER PLAYERS

Although dealing with collisions after they have occurred is important, it is preferable to prevent such collisions in the first place. We decided that the most important time for a player to avoid collisions is once it has possession of the ball. When a player is going after the ball, a collision can sometimes actually be helpful (running into an opponent that has control of the ball, for example). Until this point, our strategy had essentially been to get to the ball, orbit to face the opponent's goal, and then shoot. With the ability to avoid other players, we would be able to incorporate dribbling into our strategy. First, though, I had to improve our robots' detection of other players.

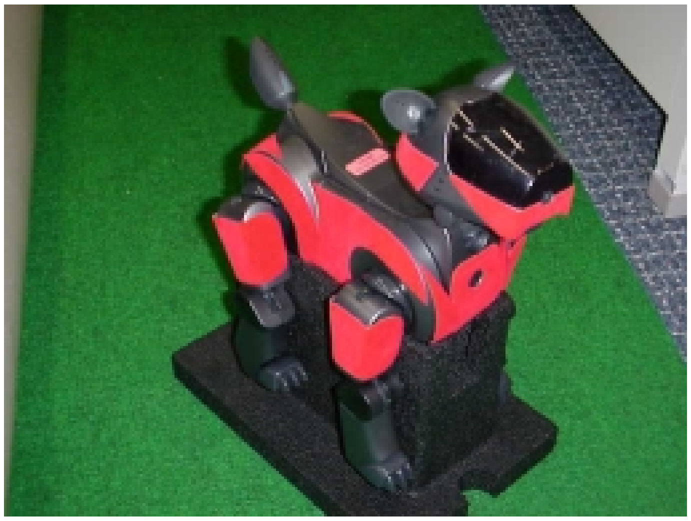
Figure 4: The players' jerseys (from [4]).

4.1 Advances in the Foe and Pal Detection Systems

The existing foe and pal detection systems were very elementary. Our robots had the ability to detect only a single player from each team - whichever player was closest at the moment. Even this limited detection was not very accurate, as no careful calibration had been done mapping the size of a robot blob to its distance. I have worked on the abilities to both store the positions of multiple players from each team and to make more accurate estimates of players' positions.

4.1.1 Storing Locations of Multiple Players

Planning a path that avoided all the other nearby players would require storing the locations of all other players on the field - the three foes and two other pals. This was complicated by the fact that the robots are not one continuous color. They are identified by blue or red "jerseys," multiple stickers attached to given parts of their bodies (Figure 4). Thus two red blobs could be part of the same player, or they could represent two different players.

To distinguish between these two possibilities, I wrote a function to determine whether a blue or red blob is part of the same player represented by any other, larger blob of the same color. This function compares the distance between two blobs with the size of the larger blob, determined by taking the average of the width and height of the blob.[6] After trying a few numbers, I found it worked best to assume two blobs are part of the same player if the distance between them is less than three times the size of the larger blob.

[6] I did not use the area of the blob because I wanted size to be a linear function of the dimensions of the blob. If a player moves twice as far away, then the lengths and widths of its component blobs will both be cut approximately in half. The distances between the different blobs of the player will also be approximately halved, not quartered, as would be suggested by basing size on the area of a blob.

Using a factor of three errs on the side of counting two blobs as part of the same player too often. Erring in this direction is preferable, as it is usually better to ignore a player that is actually present than to see a "ghost" player. Ignoring a present player means ignoring only those farther away, since blobs are considered in order of size; seeing a "ghost" player, however, may cause the robot to forget the position of a valid player, replacing it with the position of this non-existent player.

Once the positions of all visible players are determined (as discussed below), the observer must determine which player in memory corresponds to each player in the current view. This allows a robot to remember the position of players that it does not currently see, while updating the positions of those it does see. The algorithm that determines this matching is based on a similar algorithm presented by Veloso et al. [5]. For a player *A* in the current view and a player *B* in memory (with *B* on the same team as *A*), I define the likelihood that *A* and *B* are the same player as:

*likelihood(A,B) =(B's uncertainty) * 1/(distance from A to B)*^{3} ** 1/(angle difference between A and B)*

Uncertainty ranges from 0.5 (if *B* has just been seen) to 100 (if *B* has not been seen in a long time). Thus, it is more likely that they are the same player if we haven't seen *B* in a while, and also more likely if *A* and *B* are close. Then, for each player *A* in the current view, the player *B* in memory that gives the highest *likelihood(A,B)* (and has not already been chosen for another player in the current view) is replaced in memory by *A*.

I had first tried multiplying *uncertainty* by *1/distance* rather than *1/distance*^{3}, but found that this frequently caused a single foe to fill up all three foe slots, if that was the only foe the player was currently seeing. As the other foes' uncertainties rose slightly, uncertainty became more important than distance moved, and so the robot would replace these less certain foes. I also added an extra factor of *1/(angle difference)* since the angle to a player tends to be more accurate than distance, though it is not clear that this helps. Although I have not tested this algorithm extensively, it seems to perform adequately. It at least usually prevents one player from occupying multiple slots, which was our major concern.

4.1.2 Improvements in the Player Position Estimates

As with the ball, the angle to a player is determined using a combination of the robot's head position and the location of the player in the camera image, and the distance is in most cases determined using a calibration table mapping the number of pixels in the player's image to its distance. As this calibration had not yet been done, I had to build the table before proceeding.

One difficulty in doing the calibration was the robots' asymmetry. The width of a player differs depending on the angle at which it is viewed, but its height (and, more importantly, the height of its jersey) remains constant. I therefore chose to do the calibration based only on height. After some experimentation, however, I discovered that even the height differs slightly depending on the viewing angle. I had originally done the calibration using a side view, but found that when viewed head-on, the robots appeared slightly smaller. Because a robot’s most common view of another player is a head-on view, and because it is safer to err on the side of assuming that another player is closer than it actually is, I redid the calibrations using a front view.

When multiple blobs were part of the same player, I could determine the player’s position using either the combined sizes and centers or just the size and center of the largest blob. Although I originally used the combined sizes and centers, I found that this added more uncertainty to the position estimates. When a player moved far enough away that its head stickers became too small to detect, its height would suddenly drop in half. In addition, combining the blobs could yield inaccurate position estimates if the algorithm determining whether two blobs were part of the same player did not work reliably. I could not see any major disadvantage to using only the largest blob of each player to determine its position, and doing so seemed safer - and made the computations simpler. Thus this is the method I chose.

Because it is often more difficult to obtain an accurate distance estimate for a player than for the ball, I chose to use the reading from the infrared distance sensor whenever a player occupies the center of the robot’s vision and the sensor gives a reading under 40 cm, its maximum reliable range. Finally, as with the ball, I wrote an algorithm to update the relative positions of all other players based on a robot’s own movement.

4.2 Artificial Potential Fields

Dribbling the ball around other players is significantly more complex than simply avoiding a single obstacle. First, multiple obstacles must all be avoided, if possible. Second, these obstacles are moving. Finally, the player with the ball must make its way to the goal at the same time that it avoids these obstacles. The technique of artificial potential fields, pioneered by Khatib [6], was designed for just these conditions; it can even handle moving obstacles about as well as stationary ones.

The idea behind artificial potential fields is that the goal exerts an attractive force on the robot, while obstacles exert repulsive forces. The force exerted by the goal is directly proportional to the robot’s distance from it; this force vector is given by:

$$F_{goal} = -k_1 * (x - x_{goal}),$$

where k_1 is a positive constant, x is the position of the robot, and x_{goal} is the position of the goal.

The force exerted by an obstacle decreases as distance from the obstacle increases, and falls to zero beyond some given distance. Specifically, the force is proportional to the inverse of the cube of its distance; thus the force drops rapidly as the distance to an obstacle increases. For an obstacle modeled as a point, the repulsive force vector is given by:

$$F_{obstacle} = k_2 * (1/(x - x_{obstacle}) - 1/x_0) * 1/(x - x_{obstacle})^2 * grad(x - x_{obstacle}),$$

where k_2 is a positive constant, x is the position of the robot, $x_{obstacle}$ is the position of the obstacle, and x_0 is the greatest distance at which an obstacle exerts a force. If $x > x_0$, $F_{obstacle}$ is defined to be 0 [6, 7].

By summing the forces acting on the robot at a given point, it is possible to find the total force vector; the robot should then move in the direction of this vector. This will effectively steer the robot toward the goal and away from the obstacles. A major advantage of this method is that it allows real-time planning, because calculating the desired direction of travel at any moment is not very computationally expensive. Furthermore, the method lends itself to applications with moving obstacles [6]. Rather than planning the entire path at once, only the current velocity vector is calculated. Thus, in each iteration, only a few calculations are needed, and they can use the most recent observations of the obstacles' positions.

4.3 Application of Artificial Potential Fields to Obstacle Avoidance in RoboCup

Before attempting to implement the method of artificial potential fields on the robots, I wrote a simple computer simulation that allowed me to observe the effects of varying the different constants in the equations. It was clear, though, that this model would not carry over well to the actual robots, because of the robots' noisy sensors, complex shape, and delayed movements. Thus I did not do much experimentation with it before moving to an implementation on the robots. The simulation made apparent, however, that I would have to modify the theoretical model of artificial potential fields in order for it to work well in practice on the robots. This became even clearer when I began to implement the algorithm on the robots.

I chose to model all the other players as point obstacles and the dribbling robot as a point as well. This was partly to simplify calculations and partly because the robot's limited vision does not allow it to determine the orientation of a player, so it could not accurately model other players as three-dimensional, or even two-dimensional, obstacles. I also model the side walls as obstacles that always exert a force perpendicular to themselves, though it is not clear that the repulsive force from the walls is necessary or even helpful, as I have not performed much testing of changes in this force.

4.3.1 Basic Modifications of Theoretical Force Equations

My original model of the forces was similar to the theoretical model. The goal's force was proportional to the robot's distance from it, and obstacle forces were inversely proportional to the cube of their distance. At first, I did not have a maximum distance for the obstacles' forces, but I later incorporated a maximum for a number of reasons. First, with moving obstacles, the position of a distant obstacle at any given moment may be very different from its position when the dribbler gets closer to it. Second, the robot will sometimes see small player-colored blobs that are not actually part of any player; I wanted to prevent these blobs from affecting the robots' movements. Finally, as Khatib mentions, preventing forces from such distant obstacles will yield a more stable path [6].

In my initial tests, I had difficulty finding constants (k_1 and k_2 in the above equations) that caused a large enough repulsive force from close obstacles, while not causing too large a force from slightly more distant obstacles. Because I wanted to do more than simply prevent collisions - I wanted to plan a relatively smooth and collision-free path to the goal - having obstacles' repulsive force drop off so rapidly as distance increased did not seem ideal.

I first tried making the obstacles' force inversely proportional to the square (rather than the cube) of their distance. This helped, but I still experienced the same difficulty to some degree. Thus I made an obstacle's force inversely proportional simply to its distance, and this has helped a great deal. Rather than getting close to an obstacle and then suddenly moving to one side to avoid it, the robots now will start to react when another player is more distant, and will travel in a smoother path to the goal. The equation for the repulsive force of an obstacle now became:

$$F_{obstacle} = k_2 * (1/(x - x_{obstacle}) - 1/x_0) * grad(x - x_{obstacle}).$$

Another difficulty in choosing the constants arose from the fact that the goal's force differed depending on its distance. I did not see any reason that this should be the case, and so made the magnitude of the force from the goal constant (always 1). This allows the robots' reactions to obstacles to be the same regardless of their distance from the goal, which has greatly simplified other calibrations.

Even after these adjustments, the robots' behavior was not ideal. They would frequently steer farther to the side of an obstacle than was necessary. Because other players exert repulsive forces on the dribbling robot whether they are in front of it, to its side, or behind it, they continue to repel the dribbler even after a collision is no longer possible. To remedy this, I multiply the players' repulsive forces by:

$$cos(angle\ to\ player - angle\ to\ goal).$$

If this value is negative, the player exerts no repulsive force. Thus, a player directly between the dribbler and the goal will exert the full repulsive force, but a player to the dribbler's side, where *(angle to player – angle to goal)* = 90°, will not exert any force. Similarly, a player behind the dribbler exerts no repulsive force. This added factor has greatly improved the robots' performance, allowing them to avoid players blocking their path to the goal while causing them to be unaffected by players with which there is no threat of collision.

4.3.2 Determination of Constants

Determining the best values of the important constants was mostly a matter of trial and error. The important constants, as well as the values I found gave the best performance, are summarized below:

- **Time between recalculations:** Rather than continuously recalculating the best path, the robots only perform such recalculations at discrete time intervals. Since it takes about 1/2 second for them to take a single step, I chose a time interval of 1/2 second between recalculations.

- **Movement per step:** This constant determines how far the robot should move with each step; the amount of movement is constant regardless of the magnitude of the force acting on the robot. I started with a value of 4 cm per step, but this tended to cause a dribbler to accidentally kick the ball away too often. I lowered this to 3.5 cm per step, which allows the robots to maintain control of the ball most of the time.

- **Maximum obstacle distance (x_0 in above equation):** This is the distance beyond which an obstacle exerts no force. There are actually two values for this constant, one for players and one for walls. For players, a value of 1 meter works well. For stationary obstacles, a smaller constant is sufficient, but since moving players tend to be moving toward the player with the ball, the dribbler has to react sooner. For walls I chose a value of 35 cm, though I have not tested the effect of changing this value.

Obstacle force multiplier (k_2 in above equation): Again, there are two values for this constant, one for players and one for walls. For players, a value of 800 seems to work best. Since distances are measured in mm, and the maximum distance is 1000 mm, this means that an obstacle about 45 cm away exerts a repulsive force equal to the attractive force of the goal. As with the maximum distance, a smaller value of this constant works for stationary obstacles, but this larger repulsive force is needed when the other players are moving toward the dribbler. For the walls, I am using a value of 200, so the walls exert significantly less force than other players; again, though, I have not tried changing this value.

4.3.3 Other Modifications

In both the simulation and with the actual robots the total force vector sometimes actually pushed the robot away from the goal - not a good situation when it is trying to get the ball away from its own goal and toward the opponent's goal. This was due largely to the well-known problem of "local minima" in artificial potential fields [7]. There may exist a non-goal point to which the robot is always forced back, regardless of its direction of movement. For example, this may happen if an obstacle is directly between the robot and the goal The forces from the goal and the obstacle will act in opposite directions, and the robot will stop at the point where these two forces are equal. Since there are no forces pushing the robot around the obstacle, it will never leave this point.

I solved this problem by never allowing the robot to travel in a direction more than 70° away from the direction of the goal. Originally I just reduced its angle of movement to 70° away from the goal in the appropriate direction (i.e., left or right). Later, though, I found that this did not entirely solve the problem. The robot would sometimes bounce back and forth between moving left and right. It seemed that this would usually happen when the total force from the obstacles was pushing the robot mostly away from the goal - i.e., the difference between the goal's attractive force and the obstacles' total repulsive force was more than, say, 135°. Thus, when the total force is pushing the robot more than 70° from the goal, and the difference between the goal's force and the obstacles' total force is greater than 135°, I have the robot move in the same direction (left or right) it was moving before, at an angle of 70° away from the goal.

This solved the problem of indecision in most cases, but not all the time. I tried having the robots continue to move in the same direction more often, but this made their reaction time to moving obstacles too slow. I therefore decided to leave in place the algorithm described above.

After ensuring that the robot is dribbling in the general direction of the opponent's goal, I do another check to make sure it has enough forward velocity.[7] I found that if the robot moves more than about 45° to its side, it frequently loses control of the ball. Thus, if necessary, the robot's angle of movement is further adjusted to be no more than 45° from straight ahead. If there is another player directly in front of the robot, however, moving forwards too much tends to interlock the robots' legs; in this case, I allow an angle of up to 55° from straight ahead.

Finally, I found that, when another robot was close to the dribbler and had its head down (looking at the ball), the dribbler saw this robot as being much farther than it actually was. This was due to the player's black head partially obstructing its colored jersey. To allow the robots to detect this situation, I added an obstacle variable based on the reading from the infrared distance sensor. If the infrared distance sensor is giving a significantly closer reading to an obstacle than the distance estimates to the various players, the robot will use the infrared reading instead of one of the player readings.

4.3.4 Performance

The method of artificial potential fields, modified as discussed above, has given good results in controlled tests, but I have not yet tested it extensively in actual game play. A dribbling robot is able to consistently avoid stationary players. When there is one other moving player, its performance is also good, whether it starts dribbling when the other player is near or far - even when the other player is moving directly toward it. When there

[7] Dribbling toward the opponent's goal and dribbling forwards are not necessarily equivalent, if the robot is not pointed directly toward the goal.

are, additionally, one or more stationary obstacles, it performs well much of the time. But in some situations, such as when the moving player is to one side, and a stationary player to another, it does not perform as well. I have not yet performed many tests with more than one other moving player.

Overall, the results from these tests are pleasing, and we plan to incorporate this dribbling strategy into the final code for the RoboCup tournament in early August.

5. RECOMMENDATIONS

The existing sensor systems, while working adequately, could still use some improvement, especially in the calculation of objects' distances. The fix for this may be as simple as using the high resolution camera rather than the low resolution camera. A more accurate determination of the ball's distance would be especially helpful, as it would allow a more stable reading of its velocity. This in turn could allow more advanced planning based on the ball's velocity, such as different goalie reactions depending on the speed of the ball. We have begun work in this area, but there is still much that can be done.

The obstacle avoidance algorithm in place now works fairly well, but could still be fine-tuned. More work could be done on solving the problem of local minima, thus dealing with the robots' problem of indecision. Finally, information about the location of a player's teammates could be used more extensively, for applications such as passing. We are just beginning to do work in this area.

6. CONCLUSIONS

I have improved the AIBOs' ball and player detection systems in a number of ways. I have slightly improved the accuracy of the ball's position estimation, and have added the ability to calculate the ball's velocity. I have also significantly increased the accuracy of the detection of other players. In addition, I implemented a mechanism for storing the locations of all other players on the field.

Using these advances in the sensor systems, I have developed algorithms for avoiding collisions and for recovering from collisions when they do occur. I use the ball's relative velocity to determine when a robot is stuck, and use a simple backing up and sidestepping method to free it from the collision. Finally, using a modified artificial potential field method, I have given the robots the ability to dribble the ball to the goal, avoiding the other players on the field.

7. ACKNOWLEDGMENTS

I would like to thank Professor Jim Ostrowski for his suggestions and help with all aspects of this project. I am also grateful to the graduate students working on the project, Sachin Chitta and Aveek Das, for their suggestions and help developing parts of the code with which I was less familiar. Janice Fisher provided invaluable support in her editing of this paper. Finally, I would like to thank the National Science Foundation for their NSF-REU grant that made this research possible, and Microsoft Corporation for their financial support.

8. REFERENCES

1. H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa, RoboCup: The Robot World Cup Initiative, Proc. IJCAI-95 Workshop on Entertainment and AI/ALife, Montreal, 1995, pp. 19-24.

2. K. McIsaac, A. Das, S. Chitta, J. Neiling, A. Give'on, C. Klose, I. Drexler, W. Chu, J. Lyons, and J. Ostrowski, Report to Sony for UPennalizers 2000, *Unpublished Manuscript*, GRASP Laboratory, University of Pennsylvania, 2001.

3. Sony, *AIBO Homepage: Main Specifications*, http://www.aibo.com/ers_210/ers_210_spec.html, accessed July 22, 2001.

4. Sony, *OPEN-R Support for RoboCup-2001 Seattle: Team Markers 2001*, https://www.openr.org/page1_2001/uniform/uniform2001.html, accessed July 24, 2001.

5. M. Veloso, M. Bowling, S. Achim, K. Han, and P. Stone, The CMUnited-98 Champion Small Robot Team, in M. Asada and H. Kitano, eds., *RoboCup-98: Robot Soccer World Cup II*, Springer Verlag, Berlin, 1999.

6. O. Khatib, Real-Time Obstacle Avoidance for Manipulators and Mobile Robots, *Int. J. of Robot Res.*, 5(1) (Spring 1986) 90-98.

7. M. Gill and A. Zomaya, *Obstacle Avoidance in Multi-Robot Systems*, World Scientific, Singapore, 1998, pp. 115-120.