UT Austin Villa 2011
3D Simulation Team Report

Patrick MacAlpine, Daniel Urieli, Samuel Barrett,
Shivaram Kalyanakrishnan[1], Francisco Barrera,
Adrian Lopez-Mobilia, Nicolae Ştiurcă[2], Victor Vu, Peter Stone
Department of Computer Science
The University of Texas at Austin, Austin, TX 78701, USA
{patmac, urieli, sbarrett, shivaram, tank225,
alomo01, nstiurca, diragjie, pstone}@cs.utexas.edu

Technical Report AI11-10

[1]At the time of publication, Shivaram Kalyanakrishnan is affiliated with Yahoo! Labs Bangalore, Bengaluru 560071, India (e-mail: shivaram@yahoo-inc.com).
[2]At the time of publication, Nicolae Ştiurcă is affiliated with the University of Pennsylvania, Philadelphia, PA 19104, USA (e-mail: nstiurca@seas.upenn.edu).

Abstract

The RoboCup 3D simulation league is an international competition in which autonomous simulated humanoid robots play soccer against each other in a physically realistic environment. This report presents the architecture, design decisions, and components of the UT Austin Villa 2011 RoboCup 3D simulation league team. Key components include an omnidirectional walk engine and associated walk parameter optimization framework, an inverse kinematics based kicking architecture, and a dynamic role and formation positioning system. UT Austin Villa won the RoboCup 2011 3D simulation competition, consisting of 22 teams from 12 different countries, in convincing fashion by winning all 24 games it played. During the course of the competition the team scored 136 goals while conceding none.

Contents

1 Introduction

The UT Austin Villa RoboCup 3D simulation team was formed in 2007 and at the time consisted of only one graduate student and one professor: Shivaram Kalyanakrishnan and Peter Stone. The team competed in both the 2007 and 2008 RoboCup competitions, and although it failed to win a match or score a goal in either of these, considerable headway was made in building up an agent architecture and engineering necessary primitives such as walking and kicking a ball. After taking the 2009 year off, the team increased in size to include four graduate students, and had a respectable showing in the 2010 competition scoring 11 goals and winning four matches while finishing just outside the top eight. In 2011 the size of the team expanded again with the addition of six students recruited from an undergraduate course on autonomous multiagent systems taught by professor Peter Stone. Bolstered by a larger team, and building on the success and lessons learned from the 2010 competition, the team improved substantially and won the 2011 competition in convincing fashion by winning all 24 games it played. During the course of the competition the team scored 136 goals while conceding none.[1]

This report documents the architecture, design decisions, and components that are part of the 2011 UT Austin Villa world champion RoboCup 3D simulation league team. A significant portion of the content presented has been accepted for conference publication. Readers are urged to cite [10] in lieu of this report when appropriate.

The rest of the report is structured as follows. Section 2 provides a description of the RoboCup 3D simulation domain. In Section 3 we describe our agent's architecture. Section 4 describes our perception system and Section 5 discusses localization. Section 6 details our communication system. In Section 7 we describe our skill description language file framework. Section 8 covers fall detection and recovery. Section 9 discusses our goalie. Section 10 presents our omnidirectional walk engine and associated walk parameter optimization framework. In Section 11 we explain our dynamic role and positioning system. Section 12 discusses general locomotion in the field. In Section 13 we describe our inverse kinematics based kicking architecture. Section 14 discusses our agent's behavior for penalty kicks. Competition results are given in Section 15, and Section 16 summarizes.

2 Domain Description

The RoboCup 3D simulation environment is based on SimSpark [3], a generic physical multiagent system simulator. SimSpark uses the Open Dynamics Engine [2] (ODE) library for its realistic simulation of rigid body dynamics with collision detection and friction. ODE also provides support for the modeling of advanced motorized hinge joints used in the humanoid agents.

The robot agents in the simulation are homogeneous and are modeled after the Aldebaran Nao robot [1], which has a height of about 57 cm, and a mass of 4.5 kg. The agents interact with the simulator by sending torque commands and

[1]More information about the UT Austin Villa team, as well as video highlights from the competition, can be found at the team's website: http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/

Figure 1: A screenshot of the Nao humanoid robot (left), and a view of the soccer field during a 9 versus 9 game (right).

receiving perceptual information. Each robot has 22 degrees of freedom: six in each leg, four in each arm, and two in the neck. In order to monitor and control its hinge joints, an agent is equipped with joint perceptors and effectors. Joint perceptors provide the agent with noise-free angular measurements every simulation cycle (20 ms), while joint effectors allow the agent to specify the torque and direction in which to move a joint. Although there is no intentional noise in actuation, there is slight actuation noise that results from approximations in the physics engine and the need to constrain computations to be performed in real-time. Visual information about the environment is given to an agent every third simulation cycle (60 ms) through noisy measurements of the distance and angle to objects within a restricted vision cone (120°). Agents are also outfitted with noisy accelerometer and gyroscope perceptors, as well as force resistance perceptors on the sole of each foot. Additionally, agents can communicate with each other every other simulation cycle (40 ms) by sending messages limited to 20 bytes. Figure 1 shows a visualization of the Nao robot and the soccer field during a game.

3 Agent Architecture

The UT Austin Villa agent receives visual sensory information from the environment which provides distances and angles to different objects on the field. It is relatively straightforward to build a world model by converting this information about the objects into Cartesian coordinates. This of course requires the robot to be able to localize itself for which the agent uses a particle filter (discussed in Section 5). In addition to the vision perceptor, the agent also uses its accelerometer readings to determine if it has fallen (discussed in Section 8) and employs its auditory channels for communication (discussed in Section 6).

Once a world model is built (discussed in Section 4), the agent's control module is invoked. Figure 2 provides a schematic view of the control architecture of the UT Austin Villa humanoid soccer agent.

At the lowest level, the humanoid is controlled by specifying torques to each of its joints. This is implemented through PID controllers for each joint, which take as input the desired angle of the joint and compute the appropriate torque.

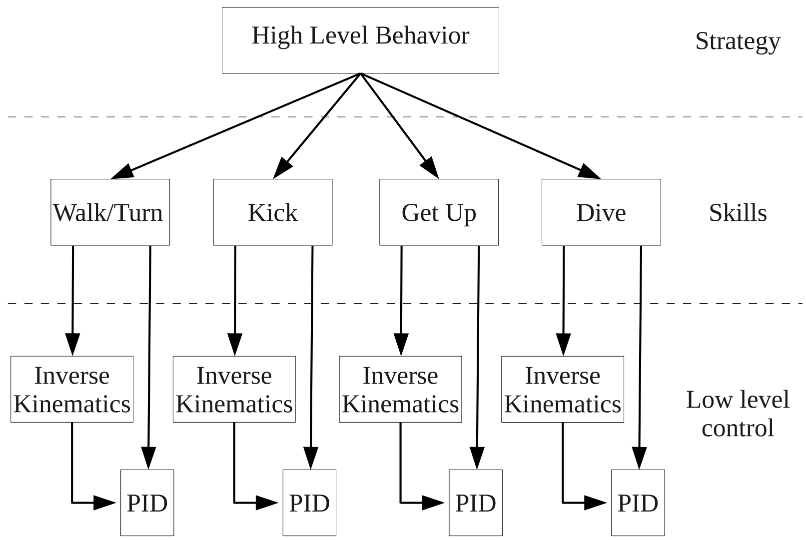

Figure 2: Schematic view of UT Austin Villa agent control architecture.

Further, the agent uses routines describing inverse kinematics for the arms and legs. Given a target position and pose for the hand or the foot, the inverse kinematics routine uses trigonometry to calculate the angles for the different joints along the arm or the leg to achieve the specified target, if at all possible.

The PID control and inverse kinematics routines are used as primitives to describe the agent's skills. In order to determine the appropriate joint angle sequences for walking and turning, the agent utilizes an omnidirectional walk engine which is described in Section 10. When invoking the kicking skill, the agent uses inverse kinematics to control the kicking foot such that it follows an appropriate trajectory through the ball as described in Section 13. Two other useful skills for the robot are diving (by the goalie to block a ball) and getting up from a fallen position. Both diving and getting up are accomplished through a programmed sequence of poses and specified joint angles as discussed in Section 7.

Because the team's emphasis was mainly on learning robust and stable low-level skills, the high-level team strategy is relatively straightforward. The player closest to the ball is instructed to go to it while other field player agents dynamically choose target positions on the field based on predefined formations, as described in Section 11, that are dependent on the current state of the game. For example, if a teammate is dribbling the ball, one agent positions itself slightly behind the dribbler so that it is ready to continue with the ball if its teammate falls over. The goalie (described in Section 9) is instructed to stand a little in front of its goal and, using a Kalman filter to track the ball, attempts to dive and stop the ball if it comes near.

4 Perception System

The agent receives noisy vision percepts from the server every 0.06 seconds. Vision percepts in polar coordinates are received for all objects that are within a 120° view cone origined at the robot's head. In this section we describe how the agent moves its head to monitor objects (Section 4.1), how the agent maintains objects' positions in memory in a *WorldObjects* array (Section 4.2), and how we use visual memory to enhance our basic system to handle unseen objects whenever possible (Section 4.3).

4.1 Head Movement

The agent continually pans its head from side to side in order to monitor all objects on the field. This consists of a repeating cycle, eight seconds in duration, during which the agent adjusts the pan of its head every two seconds by starting out looking straight ahead, turning its head 120° to the left, returning to looking straight ahead, turning its head 120° to the right, and then returning back to looking straight ahead again. As the agent has a 120° view cone this movement allows for a full 360° vision sweep of the field. The only time the agent does not spin its head around for 360° coverage of the field is when it is focusing on the ball. When focusing on the ball the agent still pans its head left and right, but centers its view on the ball while moving its view 30° to the left and right of the ball. The goalie, described in Section 9, always stays focused on the ball in order to accurately track it. Field player agents focus on the ball when they are within a meter of it. This is necessary for dribbling (Section 12.6) and kicking (Section 13). The agent also keeps its head tilted at a 45° angle downward providing continual vision both 15° above the agent's camera and 15° behind the agent's feet.

4.2 World Objects

The basic structure for holding an object is:

```
struct WorldObject {
  id; // unique id for all objects that can ever be seen
  visionInformation; // polar coordinates from robot to object
  position; // global, cartesian position
  isCurrentlySeen; // is the object seen in the current cycle
  isValid; // false if looking at position but do not see object
};
```

In the RoboCup 3D simulation domain the set of objects is fixed: teammates, opponents, ball, goal posts, and field corners. Therefore, objects are maintained in an array of WorldObjects, indexed by the object id. Whenever vision percepts are sent, the polar coordinates are stored in the *visionInformation* field, and are translated to global position after the agent updates its localization. In addition, the vision information of fixed objects, like goal posts and field corners, is used by the localization system to update the agent's belief about its location (as is described in Section 5).

One point that is noteworthy is that the vision percepts arrive in polar coordinates with respect to the robot's head, which continually pans left and right, to increase the robot's field of view. This might create a problem: it is likely that at the times of recording vision information of two different objects, the head is in two different positions with respect to the torso. In addition, the robot's head is slightly bent downwards, so that even if the head is looking straight ahead, the recorded angles to objects are still different than their angles with respect to the torso. Therefore, as a preprocessing step, vision data is transformed using several matrix multiplications to be with respect to the robot's torso. This is done to create a common frame of reference during the translation of vision information to global field coordinates.

4.3 Visual Memory

When an object, such as another agent or ball, is no longer in the field of view of the robot it is assumed that the object remains at the same position it was last seen. The location of the object is updated once it is seen again or if another agent communicates the location of the object (see Section 6). If the stored position of the object is in the agent's current field of vision, but the agent no longer sees the object there, the location of the object is marked as being no longer valid (*isValid* field of WorldObject struct shown in Section 4.2 is set to `false`).

5 Localization

Localization is done using a particle filter. Our particle filter, also known as Monte Carlo localization [6], was originally written for the real NAO robot of our SPL team and was adapted to the simulation environment. In our particle filter, 1000 particles are updated every cycle, where a particle is an (x, y, θ) estimate of an agent's pose, and a probability assigned to this estimate. The agent estimates its (x, y) position and θ orientation as weighted averages of the particles' positions and orientations, respectively, weighted by the particles' weights. Each time the agent receives a set of vision percepts, the particle filter performs the following three steps: (1) update particles from odometry, (2) update particles from landmark observations, and (3) resample particles.

Update Particles from Odometry In this step, odometry information in the form of $(\delta x, \delta y, \delta\theta)$ is extracted from the walk engine. Then, for each particle, the following update is performed:

$$(x, y, \theta) := (x, y, \theta) + (a \cdot \Delta x, b \cdot \Delta y, c \cdot \Delta\theta)$$

where a, b, and c are factors we tuned by hand, using manual measurements.

Update Particles from Landmark Observations This step, in which particles' weights are updated from landmark observations, is done only when at least one landmark is seen. While in general at least two landmarks are needed for position estimation, when the agent is roughly localized, it is possible to decrease its estimation error even with one landmark. Our particle filter is able to take advantage of this and updates its particles whenever at least one landmark is seen. In this step we use the standard method of initializing all particles to the same weights, and then multiplying their weight by the probabilities they assign to the currently seen landmarks: the difference between the expected measurement and the actual measurement is input to a Gaussian distribution, which in turn determines the probability of this difference. We have two such Gaussian distributions, one for the distance error, and one for the orientation error. Both of the Gaussians' standard deviations were hand tuned using a trial and error process. Note that as the multiplied probabilities can get small, we use the log-probabilities in our numerical calculations.

Resample Particles In this step we use the new weights computed in the previous step as a new distribution over the particles, and resample from it 1000 new particles. More accurately, 5% of the particles are sampled completely randomly in the field's area, to handle a kidnapped agent scenario. This step is done only when at least one landmark is seen, such that particles' weights were updated in the previous step. If no landmarks are seen, the agent just uses its odometry updates and no resampling is done, to avoid inserting additional noise. After particles are resampled, we insert a small amount of noise to each one of them by slightly moving their position and orientation according to a random walk.

6 Communication System

As described in Section 2, our soccer agents only receive noisy and restricted perceptual information. Consequently none of the agents possesses complete and perfect state information about the world. In such a scenario, inter-agent communication can significantly add to each agent's knowledge about the world and improve decision making.

The 3D simulator provides an "audio" channel for agents to communicate. An agent may broadcast such a SAY message once every 2 cycles (40 milliseconds); agents receive HEAR messages corresponding to all the SAY messages sent in the previous cycle. The HEAR messages do not come tagged with information identifying the sender, and so we find it necessary to send identifying information within the message itself.

The 3D simulation server allows for communication messages of size 20 ASCII characters (with a small number of ASCII characters disallowed). The UT Austin Villa agent conservatively uses a dictionary of only 64 ASCII characters (out of roughly 250 that are allowed to be transmitted): the number of distinct patterns we can therefore communicate is $64^{20} = 2^{120}$. Below we describe the rationing of the 120 bits at our disposal for communicating different types of information. Table 1 breaks down the number of bits allocated to each piece of information communicated.

A unique 16-bit pattern signature identifies whether the sender is from UT Austin Villa, and a further check of the server time (in cycles) and the sender's side are used to virtually eliminate the possibility that our agent gets confused by messages being sent by another team or, during self play, by our own agent on the opponent team. The body of the message contains information on whether the agent has fallen down, its perceived position of itself and the ball (suitably discretized), whether the agent has seen the ball in the previous vision cycle, and whether its information is reliable (as determined by a suitable rule of whether

Field	Number of bits
Signature	16
Server time in cycles	16
Sender's side (left/right)	1
Is the sender fallen?	1
Sender's perceived ball X coordinate	10
Sender's perceived ball Y coordinate	10
Sender's perceived self X coordinate	10
Sender's perceived self Y coordinate	10
Is sender seeing ball?	1
Can sender's information be trusted?	1
Role assignment vector	36 (9 * 4)

Table 1: Number of bits allocated to each piece of information communicated.

or not the agent thinks it is accurately localized). In addition one of our main uses for the communication channel is for all the agents to converge on the same role-allocation vector, which determines a pairing between the agents and positions on the field. Our role allocation coordination system is described in detail in Section 11.4. Each player (which in our case is 9) is assigned a role with each role represented by a fixed number using 4 bits.

As an added layer of security for our communication system, we use a simple encryption scheme to scramble the bits in a message before it is sent out as a SAY message; a corresponding decryption scheme unscrambles the bits on receipt of HEAR messages.

7 Skill Files/Framework for Open-Loop Skills

Our agent has several open-loop skills, like getting up, goalie-diving and kicking, each of which is implemented as a periodic state machine with multiple *key frames*, where a key frame is a static pose of fixed joint positions. Key frames are separated by a waiting time that lets the joints reach their target angles. To provide us flexibility in designing and parameterizing skills, we design an intuitive skill description language that facilitates the specification of key frames and the waiting times between them. Below is an illustrative example describing the diveRight skill.

```
SKILL DIVE_RIGHT

KEYFRAME 1
reset ARM_LEFT ARM_RIGHT LEG_LEFT LEG_RIGHT end
setTarget JOINT1 $jointvalue1 JOINT2 $jointvalue2 ...
setTarget JOINT3 4.3 JOINT4 52.5
wait 0.08

KEYFRAME 2
increaseTarget JOINT1 -2 JOINT2 7 ...
setTarget JOINT3 $jointvalue3 JOINT4 (2 * $jointvalue3)
wait 0.08
.
.
```

.

As seen above, joint angle values can either be numbers or be parameterized as `$<varname>`, where `<varname>` is a variable value that can be loaded after being learned. Note that due to left-right symmetry, some of these parameters influence multiple key frames.

8 Fall Detection and Recovery

Factors such as slippage on the ground and collision with objects on the field could precipitate the fall of a humanoid robot, and indeed the success of a soccer-playing robot depends crucially on the robustness of its fall management strategy. First, the robot's propensity for fall is determined by its locomotion control: naturally the robot's susceptibility to fall increases as we program it for higher locomotion speed. We describe our walk optimization routine in detail in Section 10.3: note that we penalize robots for falling while optimizing the walk.

Despite our best efforts in avoiding falls, falling is an inevitable eventuality that needs to be dealt with efficiently. To detect that a fall has occurred (or that it is impending), we use a simple rule that thresholds the X and Y components of the robot's accelerometer reading. If indeed a fall is detected based on this thresholding rule, the robot proceeds to execute a sequence of commands as part of a "get-up" routine. We observe that if the robot's arms are stretched out at 90° to the torso, along the frontal plane, the robot necessarily falls to the ground either face-up or face-down (that is, not sideways). Using the accelerometer again to detect whether it has fallen face-up or face-down, the robot proceeds through an appropriate sequence of keyframes in each case to return to an upright position. Our get-up routine is entirely manually designed and is divided into stages. If fallen face down, the robot bends at the hips and stretches out its arms until it transfers weight to its feet, at which point it can stand up by straightening the hip angle. If fallen face up, the robot uses its arms to push its torso up, and then rocks its weight back to its feet before straightening its legs to stand. These sequences are both executed entirely in an open-loop fashion, and each lasts about 2-3 seconds. There is a small probability the get-up routine is not successful (for example, if the robot is in contact with some other object while getting up): if so, the routine is repeated.

9 Goalie

The goalie is the last line of defense and is the only agent allowed to purposely dive to try and stop a ball when the opposing team shoots on goal. Although our team was never scored on during the competition, this was in large part due to our defense as our goalie never touched the ball during the course of regular gameplay. Despite our goalie being unnecessary in winning the competition, the goalie did make a great save on a well taken shot that occurred in a match after time had expired (our agents stop going to the ball once a game ends allowing for the opposing team to freely move the ball down the field). This save, against the team with arguably the best shot in the tournament (Apollo3D), suggests that our goalie would have been a factor in the competition if needed. The following sections describe the behavior of our goalie.

9.1 Positioning

Our goalie agent is designed to stay on a line .5 meters above its own goal line and always position itself between the ball and the goal so as to minimize the maximum angle between either goal post, ball, and the goalie. As the goalie moves it is instructed to always face the ball so that it can both keep track of the current position of the ball and also be in position to dive left or right at angles perpendicular to the direction of the ball for maximum angular coverage. Should the ball enter the goal box, and the goalie is determined to be the closest agent to the ball, the goalie will assume the *onBall* role (discussed in Section 11.1) and go to the ball. Otherwise the goalie is instructed to always stay within its goal box and position itself to best be ready to block shots.

9.2 Kalman Filter

Our goalie needs to quickly and accurately respond to balls traveling in toward the goal. Because accuracy is paramount in the estimation of the ball's position, and we need a way of smoothing out noise present in observations of the ball's location, our goalie uses a Kalman filter to track the ball's position and velocity. The Kalman filter was originally implemented by our SPL team for the real Nao robot. In order to adapt the Kalman filter to the simulation environment the main thing that was required was re-tuning the filter's parameters.

9.3 Dives

We equip our goalie with a special set of diving skills in order to effectively use its body to stop a ball that is headed toward the goal. Since our goalie tracks the ball velocity with a Kalman filter (Section 9.2), these dives are invoked only when the goalie evaluates that the ball is indeed headed with a certain threshold velocity toward the goal; otherwise, the goalie merely intercepts the ball by running toward it.

The key desiderata of a dive are that the goalie lower its body to the ground as quickly as possible, and that the angular range the goalie is able to "cut off" with its dive be as large as possible. We achieve these two objectives by designing three separate types of dives for the goalie; screenshots of these dives are depicted in Figure 3. Figure 3(a) shows a "central split", which results in the goalie reaching the ground with its legs split. This dive is programmed as a sequence of keyframes mainly manipulating joints in the robot's legs. Since the keyframes have left-right symmetry, the robot remains more-or-less centered at its original position when its legs split and touch the ground. Figure 3(b) shows a slight variation, a "side-wards split" that is essentially the same as a central split, but by introducing slight asymmetry in the keyframes of the skill, results in a net displacement of the robot either to the left or the right. Both the central and the side-ward splits typically take less than 1.5 seconds to complete. The third diving skill, shown in Figure 3(c), is a more human-like lateral lunge, which accomplishes significantly larger lateral coverage than the side-wards dive. Again programmed as an open-loop sequence of keyframes, this

(a) Central split (b) Side-wards split (c) Lateral lunge

Figure 3: Screenshots of the goalie diving.

dive takes about 2.0 seconds to complete; that is, for the robot's outstretched arms to touch the ground.

The strategy controlling our goalie's dives is dependent on the Kalman filter's prediction of the ball's trajectory. Depending on the line predicted to be taken by the ball, as well as the ball's speed, a manually designed set of rules determines whether a dive is to be undertaken, and if so, which of the five available dives (central split, left-wards split, right-wards split, left lunge, right lunge) is to be deployed.

10 Walking

The biggest factor in UT Austin Villa's success at the 2011 RoboCup tournament was its very efficient walk. This section describes the walk in detail, starting with the design of the walk engine in Section 10.1. Section 10.2 describes how the walk is utilized in different game situations which serves as motivation for the procedure used to optimize walk engine parameters discussed in Section 10.3. Additional implementation details regarding the learned walk are given in Section 10.4. Game results showing the importance of the walk, and its improvement over the 2010 team's walk [12], are revealed in Section 15.

10.1 Walk Engine

The UT Austin Villa 2011 team used a double linear inverted pendulum model omnidirectional walk engine based on one that was originally designed for the real Nao robot [7]. The omnidirectional walk is crucial for allowing the robot to request continuous velocities in the forward, side, and turn directions, permitting it to approach continually changing destinations (often the ball) more smoothly and quickly than the team's previous year's set of unidirectional walks [12].

We began by re-implementing the walk for use on physical Nao robots before transferring it into simulation to compete in the RoboCup 3D simulation league. Many people in the past have used simulation environments for the purpose of prototyping real robot behaviors; but to the best of our knowledge, ours is the first work to use a real robot to prototype a behavior that was ultimately deployed in a simulator. Working first on the real robots led to some important discoveries. For example, we found that decreasing step sizes when the robot is unstable increases its chances of catching its balance. Similarly, on the robots we discovered that the delay between commands and sensed changes is significant, and this realization helped us develop a more stable walk in simulation.

The walk engine, though based closely on that of Graf et al. [7], differs in some of the details. Specifically, unlike Graf et al., we use a sigmoid function for the forward component and use proportional control to adjust the desired step sizes. Our work also differs from Graf et al. in that we optimize parameters for a walk in simulation while they do not. For the sake of completeness and to fully specify the semantics of the learned parameters, we present the full technical details of the walk in this section. Readers most interested in the optimization procedure can safely skip to Section 10.2. The walk engine uses a simple set of sinusoidal functions to create the motions of the limbs with limited feedback control. The walk engine processes desired walk velocities chosen by the behavior, chooses destinations for the feet and torso, and then uses inverse kinematics to determine the joint positions required. Finally, the PID controllers for each joint convert these positions into torque commands that are sent to the simulator.

The walk first selects a trajectory for the torso to follow, and then determines where the feet should be with respect to the torso location. We use x as the forwards dimension, y as the sideways dimension, z as the vertical dimension, and θ as rotating about the z axis. The trajectory is chosen using a double linear inverted pendulum, where the center of mass is swinging over the stance foot. In addition, as in Graf et al.'s work [7], we use the simplifying assumption that there is no double support phase, so that the velocities and positions of the center of mass must match when switching between the inverted pendulums formed by the respective stance feet.

Notation	Description
maxStep_i^*	Maximum step sizes allowed for x, y, and θ
y_{shift}^*	Side to side shift amount with no side velocity
z_{torso}^*	Height of the torso from the ground
z_{step}^*	Maximum height of the foot from the ground
f_g^*	Fraction of a phase that the swing foot spends on the ground before lifting
f_a	Fraction that the swing foot spends in the air
f_s^*	Fraction before the swing foot starts moving
f_m	Fraction that the swing foot spends moving
ϕ_{length}^*	Duration of a single step
δ^*	Factors of how fast the step sizes change
y_{sep}	Separation between the feet
x_{offset}^*	Constant offset between the torso and feet
x_{factor}^*	Factor of the step size applied to the forwards position of the torso
err_{norm}^*	Maximum COM error before the steps are slowed
err_{max}^*	Maximum COM error before all velocity reach 0

Table 2: Parameters of the walk engine with the optimized parameters starred.

We now describe the mathematical formulas that calculate the positions of the feet with respect to the torso. More than 40 parameters were used but only the most important ones are described in Table 2. Note that many, but not all of these parameters' values were optimized as described in Section 10.3.

To smooth changes in the velocities, we use a simple proportional controller to filter the requested velocities coming from the behavior module. Specifically, we calculate $\text{step}_{i,t+1} = \text{step}_{i,t} + \delta(\text{desired}_{i,t+1} - \text{step}_{i,t}) \forall i \in \{x, y, \theta\}$. In addition, the value is cropped within the maximum step sizes so that $-\text{maxStep}_i \leq \text{step}_{i,t+1} \leq \text{maxStep}_i$.

The phase is given by $\phi_{start} \leq \phi \leq \phi_{end}$, and $t = \dfrac{\phi - \phi_{start}}{\phi_{end} - \phi_{start}}$ is the current fraction through the phase. At each time step, ϕ is incremented by $\Delta\text{seconds}/\phi_{length}$, until $\phi \geq \phi_{end}$. At this point, the stance and swing feet change and ϕ is reset to ϕ_{start}. Initially, $\phi_{start} = -0.5$ and $\phi_{end} = 0.5$. However, the start and end times will change to match the previous pendulum, as given by the equations

$$
\begin{aligned}
k &= \sqrt{9806.65/z_{torso}} \\
\alpha &= 6 - \cosh(k - 0.5\phi) \\
\phi_{start} &= \begin{cases} \dfrac{\cosh^{-1}(\alpha)}{0.5k} & \text{if } \alpha \geq 1.0 \\ -0.5 & \text{otherwise} \end{cases} \\
\phi_{end} &= 0.5(\phi_{end} - \phi_{start})
\end{aligned}
$$

The stance foot remains fixed on the ground, and the swing foot is smoothly lifted and placed down, based on a cosine function. The current distance of the feet from the torso is given by

$$
\begin{aligned}
z_{frac} &= \begin{cases} 0.5(1 - \cos(2\pi \dfrac{t - f_g}{f_a})) & \text{if } f_g \leq t \leq f_a \\ 0 & \text{otherwise} \end{cases} \\
z_{stance} &= z_{torso} \\
z_{swing} &= z_{torso} - z_{step} * z_{frac}
\end{aligned}
$$

It is desirable for the robot's center of mass to steadily shift side to side, allowing it to stably lift its feet. The side to side coordinate when no side velocity is requested is given by

$$
\begin{aligned}
y_{stance} &= 0.5y_{sep} + y_{shift}(-1.5 + 0.5\cosh(0.5k\phi)) \\
y_{swing} &= y_{sep} - y_{stance}
\end{aligned}
$$

If a side velocity is requested, y_{stance} is augmented by

$$
y_{frac} = \begin{cases} 0 & \text{if } t < f_s \\ 0.5(1 + \cos(\pi \frac{t - f_s}{f_m})) & \text{if } f_s \leq t < f_s + f_m \\ 1 & \text{otherwise} \end{cases}
$$

$$
\Delta y_{stance} = \text{step}_y * y_{frac}
$$

These equations allow the y component of the feet to smoothly incorporate the desired sideways velocity while still shifting enough to remain dynamically stable over the stance foot.

Next, the forwards component is given by

$$\begin{aligned}
s &= \text{sigmoid}(10(-0.5 + \frac{t - f_s}{f_m})) \\
x_{frac} &= \begin{cases} (-0.5 - t + f_s) & \text{if } t < f_s \\ (-0.5 + s) & \text{if } f_s \le t < f_s + f_m \\ (0.5 - t + f_s + f_m) & \text{otherwise} \end{cases} \\
x_{stance} &= 0.5 - t + f_s \\
x_{swing} &= \text{step}_x * x_{frac}
\end{aligned}$$

These functions are designed to keep the robot's center of mass moving forwards steadily, while the feet quickly, but smoothly approach their destinations. Furthermore, to keep the robot's center of mass centered between the feet, there is an additional offset to the forward component of both the stance and swing feet, given by

$$\Delta x = x_{offset} + -\text{step}_x x_{factor}$$

After these calculations, all of the x and y targets are corrected for the current position of the center of mass. Finally, the requested rotation is handled by opening and closing the groin joints of the robot, rotating the foot targets. The desired angle of the groin joint is calculated by

$$\text{groin} = \begin{cases} 0 & \text{if } t < f_s \\ \frac{1}{2}\text{step}_\theta(1 - \cos(\pi\frac{t - f_s}{f_m})) & \text{if } f_s \le t < f_s + f_m \\ \text{step}_\theta & \text{otherwise} \end{cases}$$

After these targets are calculated for both the swing and stance feet with respect to the robot's torso, the inverse kinematics module calculates the joint angles necessary to place the feet at these targets. Further description of the inverse kinematic calculations is given in [7].

To improve the stability of the walk, we track the desired center of mass as calculated from the expected commands. Then, we compare this value to the sensed center of mass after handling the delay between sending commands and sensing center of mass changes of approximately 80ms. If this error is too large, it is expected that the robot is unstable, and action must be taken to prevent falling. As the robot is more stable when walking in place, we immediately reduce the step sizes by a factor of the error. In the extreme case, the robot will attempt to walk in place until it is stable. The exact calculations are given by

$$\begin{aligned}
\text{err} &= \max_i(\text{abs}(\text{com}_{expected,i} - \text{com}_{sensed,i})) \\
\text{stepFactor} &= \max(0, \min(1, \frac{\text{err} - \text{err}_{norm}}{\text{err}_{max} - \text{err}_{norm}})) \\
\text{step}_i &= \text{stepFactor} * \text{step}_i \; \forall i \in \{x, y, \theta\}
\end{aligned}$$

This solution is less than ideal, but performed effectively enough to stabilize the robot in many situations.

10.2 Walk Movement and Control of Walk Engine

Before describing the procedure for optimizing the walk parameters in Section 10.3, we provide some brief context for how the agent's walk is typically used. These details are important for motivating the optimization procedure's fitness functions.

During gameplay the agent is usually either moving to a set target position on the field or dribbling the ball toward the opponent's goal and away from the opposing team's players. Given that an omnidirectional walk engine can move in any direction as well as turn at the same time, the agent has multiple ways in which it can move toward a target. We chose the approach of both moving and turning toward a target at the same time as this allows for both quick reactions (the agent is immediately moving in the desired direction) and speed (where the bipedal robot model is faster when walking forward as opposed to strafing sideways). We validated this design decision by playing our agent against a version of itself which does not turn to face the target it is moving toward, and found our agent that turns won by an average of .7 goals across 100 games. Additionally we played our agent against a version of itself that turns in place until its orientation is such that it is able to move toward its target at maximum forward velocity, and found our agent that immediately starts moving toward its target won by an average of .3 goals across 100 games. All agents we compared used walks optimized by the process described in Section 10.3.

Dribbling the ball is a little different in that the agent needs to align behind the ball, without first running into the ball, so that it can walk straight through the ball, moving it in the desired dribble direction. When the agent circles around the ball, it always turns to face the ball so that if an opponent approaches, it can quickly walk forward to move the ball and keep it out of reach of the opponent.

10.3 Optimization of Walk Engine Parameters

As described in Section 10.1, the walk engine is parameterized using more than 40 parameters. We initialize these parameters based on our understanding of the system and by testing them on an actual Nao robot. We refer the agent that uses this walk as the *Initial* agent.

The initial parameter values result in a very slow, but stable walk. Therefore, we optimize the parameters using the CMA-ES algorithm [8], which has been successfully applied previously to a similar problem in [12]. CMA-ES is a policy search algorithm that successively generates and evaluates sets of candidates. Once CMA-ES generates a group of candidates, each candidate is evaluated with respect to a *fitness* measure. When all the candidates in the group are evaluated, the next set of candidates is generated by sampling with probability that is biased toward directions of previously successful search steps. As a parallel search algorithm, we were able to leverage the department's large cluster of high-end computers to automate and parallelize the learning. This allowed us to complete optimization runs requiring 210,000 evaluations in less than a day. This is roughly a 150 times speedup over not doing optimization runs in parallel which would have taken over 100 days to complete.

As optimizing 40 real-valued parameters can be impractical, a carefully chosen subset of 14 parameters was selected for optimization while fixing all other parameters. The chosen parameters are those that seemed likely to have the highest potential impact on the speed and stability of the robot. The 14 optimized parameters are starred in Table 2. Note that maxStep_i represents 3 parameters. Also, while f_g and f_s where chosen to be optimized, their complements f_a and f_m were just set to $(1 - f_g)$ and $(1 - f_m)$ respectively.

Similarly to a conclusion from [12], we have found that optimization works better when the agent's fitness measure is its performance on tasks that are *executed during a real game*. This stands in contrast to evaluating it on a general task such as the speed walking straight. Therefore, we break the agent's in-game behavior into a set of smaller tasks and sequentially optimize the parameters for each one of these tasks. Videos of the agent performing optimization tasks can be found online.[2]

10.3.1 Drive Ball to Goal Optimization

We start from a task called driveBallToGoal,[3] which has been used in [12]. In this task, a robot and a ball are placed on the field, and the robot must drive the ball as far as it can toward the goal within 30 simulated seconds. The fitness of a given parameter set is the distance the ball travels toward the goal during that time. The agent thus optimized, which we refer to as the *DriveBallToGoal* agent, shows remarkable improvement in the robot's performance as the distance the ball was dribbled increased by a factor of 15 over the *Initial* agent. This improvement also showed itself in actual game performance as when the *DriveBallToGoal* agent played 100 games against the *Initial* agent, it won on average by 5.54 goals with a standard error of .14.

10.3.2 Multiple Subtasks Optimization

While optimizing walk engine parameters for the driveBallToGoal task improved the agent substantially, we noticed that the agent was unstable when stopping at a target position on the field or circling around the ball to dribble. We believe the reason for this is that the driveBallToGoal task was not very representative of these situations frequently encountered in gameplay. When dribbling a ball toward the goal, the agent never stops as it often does in regular gameplay. Additionally, good runs of the driveBallToGoal task receiving a high fitness value occur when the agent perfectly dribbles the ball toward the goal without losing it. Runs in which the agent loses the ball, and is then forced to approach and circle the ball once more, receive lower fitness values and are thus given less influence in the learning process.

Go to Target Parameter Set To better account for common situations encountered in gameplay, we replaced the driveBallToGoal task in the optimization procedure with a new goToTarget subtask. This task consists of an obstacle course in which the agent tries to navigate to a variety of target positions on the field. Each target is active, one at a time for a fixed period of time, which varies from one target to the next, and the agent is rewarded based on its distance traveled toward the active target. If the agent reaches an active target, the agent receives an extra reward based on extrapolating the distance it could

[2] http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/AustinVilla3DSimulationFiles/2011/html/walk.html

[3] Note that we use three types of notation for each of driveBallToGoal, *DriveBallToGoal*, *driveBallToGoal*, to distinguish between an optimization task, an agent created by this optimization task and a parameter set. Similarly for "goToTarget", "sprint" and "initial".

- Long walks forward/backwards/left/right
- Walk in a curve
- Quick direction changes
- Stop and go forward/backwards/left/right
- Switch between moving left-to-right and right-to-left
- Quick changes of target to simulate a noisy target
- Weave back and forth at 45 degree angles
- Extreme changes of direction to check for stability
- Quick movements combined with stopping
- Quick alternating between walking left and right
- Spiral walk both clockwise and counter-clockwise

Figure 4: GoToTarget Optimization walk trajectories

have traveled given the remaining time on the target. In addition to the target positions, the agent has stop targets, where it is penalized for any distance it travels. To promote stability, the agent is given a penalty if it falls over during the optimization run. Additional details about our optimization task architecture can be found in Section 10.4.2.

In the following equations specifying the agent's rewards for targets, $Fall$ is 5 if the robot fell and 0 otherwise, d_{target} is the distance traveled in meters toward the target, and d_{moved} is the total distance moved in meters. Let t_{total} be the full duration a target is active and t_{taken} be the time taken to reach the target or t_{total} if the target is not reached.

$$\begin{aligned} reward_{target} &= d_{target}\frac{t_{total}}{t_{taken}} - Fall \\ reward_{stop} &= -d_{moved} - Fall \end{aligned}$$

The goToTarget optimization includes quick changes of target/direction for focusing on the reaction speed of the agent, as well as targets with longer durations to improve the straight line speed of the agent. The stop targets insure that the agent is able to stop quickly, while remaining stable. The trajectories that the agent follows during the optimization are described in Figure 4. After running this optimization seeded with the initial walk engine parameter values we saw another significant improvement in performance. Using the parameter set optimized for going to a target, the *GoToTarget* agent was able to beat the *DriveBallToGoal* agent by an average of 2.04 goals with a standard error of .11 across 100 games. Although the goToTarget subtask is used in the driveBallToGoal task, varying its inputs directly was more representative of the large set of potential scenarios encountered in gameplay.

Sprint Parameter Set To further improve the forward speed of the agent, we optimized a parameter set for walking straight forwards for ten seconds starting from a complete stop. The robot was able to learn parameters for walking .78 m/s compared to .64 m/s using the *goToTarget* parameter set. Unfortunately, when the robot tried to switch between the forward walk and

Table 3: Game results of agents with different walk parameter sets. Entries show the average goal difference (row − column) from 100 ten minute games. Values in parentheses are the standard error.

	Initial	DriveBallToGoal	GoToTarget
Final	8.84(.12)	2.21(.12)	.24(.08)
GoToTarget	8.82(.11)	2.04(.11)	
DriveBallToGoal	5.54(.14)		

goToTarget parameter sets it was unstable and usually fell over. This instability is due to the parameter sets being learned in isolation, resulting in them being incompatible.

To overcome this incompatibility, we ran the goToTarget subtask optimization again, but this time we fixed the *goToTarget* parameter set and learned a new parameter set. We call these parameters the *sprint* parameter set, and the agent uses them when its orientation is within 15° of its target. The *sprint* parameter set was seeded with the values from the *goToTarget* parameter set. By learning the *sprint* parameter set in conjunction with the *goToTarget* parameter set, the new *Sprint* agent was stable switching between the two parameter sets, and its speed was increased to .71 m/s. Adding the *sprint* parameter set also improved the game performance of the agent slightly; over 100 games, the *Sprint* agent was able to beat the *GoToTarget* agent by an average of .09 goals with a standard error of .07.

Positioning Parameter Set Although adding the *goToTarget* and *sprint* walk engine parameter sets improved the stability, speed, and game performance of the agent, the agent was still a little slow when positioning to dribble the ball. This slowness makes sense because the goToTarget subtask optimization emphasizes quick turns and forward walking speed while positioning around the ball involves more side-stepping to circle the ball. To account for this discrepancy, the agent learned a third parameter set which we call the *positioning* parameter set. To learn this set, we created a new driveBallToGoal2[4] optimization in which the agent is evaluated on how far it is able to dribble the ball over 15 seconds when starting from a variety of positions and orientations from the ball. The *positioning* parameter set is used when the agent is .8 meters from the ball and is seeded with the values from the *goToTarget* parameter set. Both the *goToTarget* and *sprint* parameter sets are fixed and the optimization naturally includes transitions between all three parameter sets, which constrained them to be compatible with each other. Adding the *positioning* parameter set further improved the agent's performance such that it, our *Final* agent, was able to beat the *Sprint* agent by an average of .15 goals with a standard error of .07 across 100 games. A summary of the progression in optimizing the three different walk parameter sets can be seen in Figure 5. The results reported throughout this section are summarized in Table 3.

[4] The '2' at the end of the name driveBallToGoal2 is used to differentiate it from the driveBallToGoal optimization that was used in Section 10.3.1.

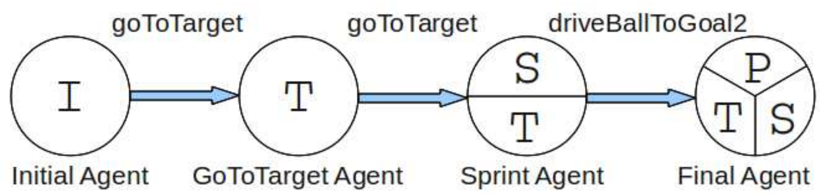

Figure 5: UT Austin Villa walk parameter optimization progression. Circles represent the set(s) of parameters used by each agent during the optimization progression while the arrows and associated labels above them indicate the optimization tasks used in learning. Parameter sets are the following: I = *initial*, T = *goToTarget*, S = *sprint*, P = *positioning*.

10.4 Additional Implementation Details

The following sections contain additional implementation details about our learned walk. These include the inputs to the walk engine (Section 10.4.1), the architecture used to build optimization tasks (Section 10.4.2), and also how the agent jogs in place before coming to a complete stop (Section 10.4.3).

10.4.1 Walk Engine Inputs

Once the agent has decided what direction it wants to walk in (`walk_direction`) and what direction it want to face (relative to its current orientation), it must give the correct inputs to the walk engine, which accepts as inputs three real numbers in the range [-1, 1]. They are the desired speed, as a percentage of the engine's maximum speed, to walk in the x and y directions and to rotate. The sign of the number determines the direction of the movement (e.g. positive X for forward and negative X for backward). Converting the desired orientation into rotation speed is simple: divide by the (admittedly arbitrary) number 180. Converting walk direction into the X/Y speeds is more tricky. We cannot simply use `sin(walk_direction)` and `cos(walk_direction)` as the Y and X speed, respectively. There are two reasons for this:

1. We generally want to walk in `walk_direction` as fast as possible. This means that either the X speed or the Y speed should equal 1.
2. The maximum X and Y speeds do not have to be the same, so we must scale the X and Y speeds accordingly.

The following formula addresses both concerns:

```
if tan(walk_direction) < (max_y_speed / max_x_speed), then
     x_speed = 1
     y_speed = tan(walk_direction) * max_x_speed / max_y_speed
otherwise,
     x_speed = tan(walk_direction) * max_y_speed / max_x_speed
     y_speed = 1
```

Under certain conditions, the agent will also want to change the walk engine parameter set it is using. The agent can do this by simply specifying the name of the desired parameter set along with the X/Y/Rotational speeds. If the walk engine is not already using the specified parameter set, it will switch its parameters values accordingly to that of the new set. This capability allows the agent to switch between different walks, optimized for different purposes, as it sees fit, instead of relying on some type of one-size-fits-all walk.

10.4.2 Optimization Task Architecture

To ease the optimization process, we build optimization runs out of a series of independent phases, called OptPhase. Each OptPhase encapsulates a logically distinct action that the agent must take, the utility function for that action, and any of the agent's observations during the execution of that phase that is used as input to the utility function. To guard against the case where the agent cannot complete the action, each OptPhase also has a maximum duration. If the agent does not complete an OptPhase's associated action within the specified duration, it simply moves on to the next OptPhase. Should the agent fall down during a phase, the next phase is not started until the agent gets up again. For optimizing the agent's walk, we primarily used the following phase types—all of which punish for falling down:

- A WaypointOptPhase, during which the agent attempts to walk to a certain coordinate and is rewarded based on how far it can walk before the phase ends. If the agent arrives at its destination before the time runs out, we try to extrapolate how far the agent would have gone if allowed to walk for the entire phase.
- A StopOptPhase, during which the agent stands still and is punished for moving. This is useful for making sure that the agent is stable, not just fast.
- A MoveOptPhase, during which the agent walks in a certain direction and is rewarded based on the distance it can travel before the phase ends.

At the beginning of an optimization run, the agent is initialized with a list of OptPhases and it simply needs to execute them all in order. Once it has gone through all of the OptPhases, it simply adds up all of the utility values for each of the individual phases and use that as the utility for the entire run. This approach allows us to quickly and easily create and experiment with different optimization strategies. For example, we can optimize for stability: by chaining together a series of short MoveOptPhases sprinkled with a number of StopOptPhases to make the agent quickly change direction, or for navigation speed: by using WaypointOptPhases to create a sort of obstacle course.

10.4.3 Stopping and Jogging In Place

Another decision on how the agent should move occurs when we want the agent to stop after reaching a desired target position on the field. We found that if the agent immediately stops and stands still after moving quickly, then stability becomes a concern with the agent often falling over due to the sudden change in motion. One way to preserve stability is to request the walk engine to have the agent jog in place instead of standing still so that the change in motion is more gradual. The drawback of jogging in place is that the added movement adds noise to the agent's localization and perception of objects around it. This is of particular concern for the goalie (discussed in Section 9) who needs very accurate measurements of the position of the ball relative to itself so that it can determine when to dive to stop the ball if the opponent attempts a shot on goal. We ended up choosing a compromise between standing and jogging in place where the agent jogs in place when stopping for .5 seconds, after which it enters a motionless standing pose.

We created and optimized the same three parameter sets used by our *Final* agent, using the process described in Section 10.3, for an agent that immediately stands in a fixed pose when asked to stop instead of jogging in place. This agent lost to our *Final* agent by an average goal difference of .64 with a standard error of .08.

11 Dynamic Role Assignment and Positioning System

While low level skills such as walking and kicking are vitally important for having a successful soccer playing agent, the agents must work together as a team in order to maximize their game performance. One often thinks of the soccer teamwork challenge as being about where the player with the ball should pass or dribble, but at least as important is where the agents position themselves when they *do not* have the ball [9]. Positioning the players in a formation requires the agents to coordinate with each other and determine where each agent should position itself on the field. In our team, players' roles are determined in three steps. First, a full team formation is computed (Section 11.1); second, each player computes the best assignment of players to roles in this formation according to its own view of the world (Section 11.3); and third, a coordination mechanism is used to choose among all players' suggestions (Section 11.4). In this section, we use the terms (player) position and (player) role interchangeably.

11.1 Formation

In general, the team formation is determined by the ball position on the field. As an example, Figure 6 depicts the different role positions of the formation and their relative offsets when the ball is at the center of the field. As can be seen in the figure, the formation can be broken up into two separate groups, an offensive and a defensive group. Within the offensive group, the role positions on the field are determined by adding a specific offset to the ball's coordinates. The *onBall* role, assigned to the player closest to the ball, is always based on where the ball is and is therefore never given an offset. On either side of the ball are two forward roles, *forwardRight* and *forwardLeft*. Directly behind the ball is a *stopper* role as well as two additional roles, *wingLeft* and *wingRight*, located behind and to either side of the ball. When the ball is near the edge of the field some of the roles' offsets from the ball are adjusted so as to prevent them from moving outside the field of play (as shown in Figures 7 and 8).

Within the defensive group there are two roles, *backLeft* and *backRight*. To determine their position on the field a line is calculated between the center of our goal and the ball. Both backs are placed along that line at specific offsets from the end line. The goalie positions itself independently of its teammates,

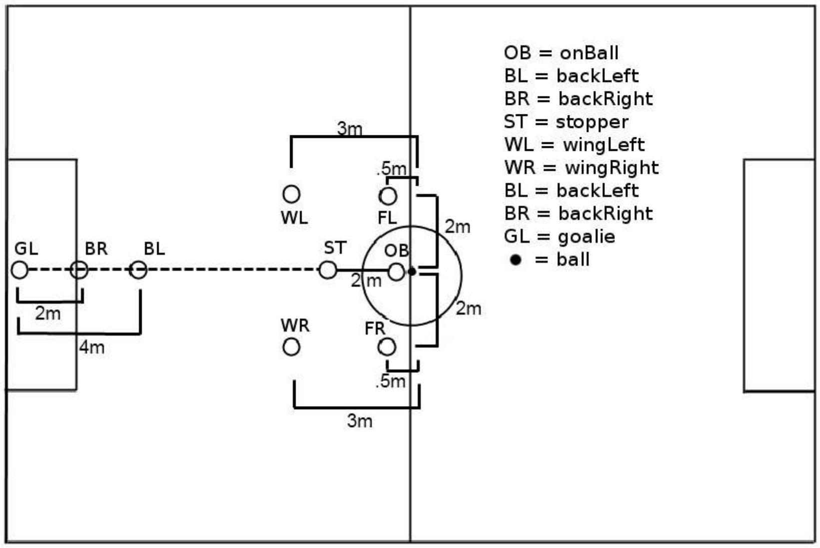

Figure 6: Formation role positions.

as described in Section 9.1, in order to always be in the best position to dive and stop a shot on goal. If the goalie assumes the *onBall* role, however, a third role is included within the defensive group, the *goalieReplacement* role. A field player assigned to the *goalieReplacement* role is told to stand in front of the center of the goal to cover for the goalie going to the ball.

11.2 Set Plays

During the course of a game there are occasional stoppages in play for events such as kickoffs, goal kicks, corner kicks, and kick-ins. When one of these events occur the team awarded a kick is given 15 seconds to move the ball while members of the other team are forced to stay a certain distance away from the ball to allow room for the kick. Our team recognizes when kicks are awarded and adjusts its team formation and behavior accordingly to account for the change in play mode.

11.2.1 Kickoff

To improve the team's field position we decided that every time our team kicks off, the agents will try and kick the ball as far into the opponent's side of the field as possible. To do this the agent on the ball randomly selects an angle, between 5 and 20 degrees, and a direction, either left or right, and positions himself around the ball accordingly. From this position the agent executes the teams most powerful kick. Using this strategy we can reliably move the ball close to six meters into the opponent's side (i.e. over half the distance to the opponent's goal).

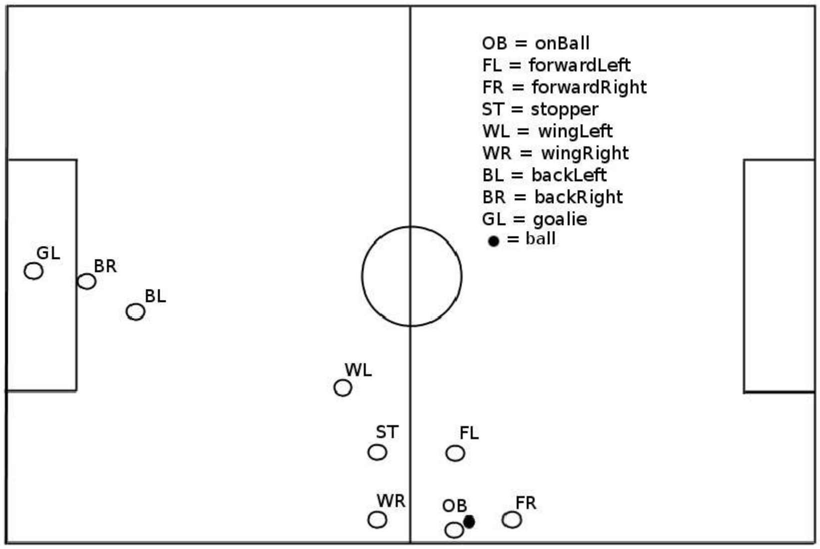

Figure 7: Shift Near Sideline

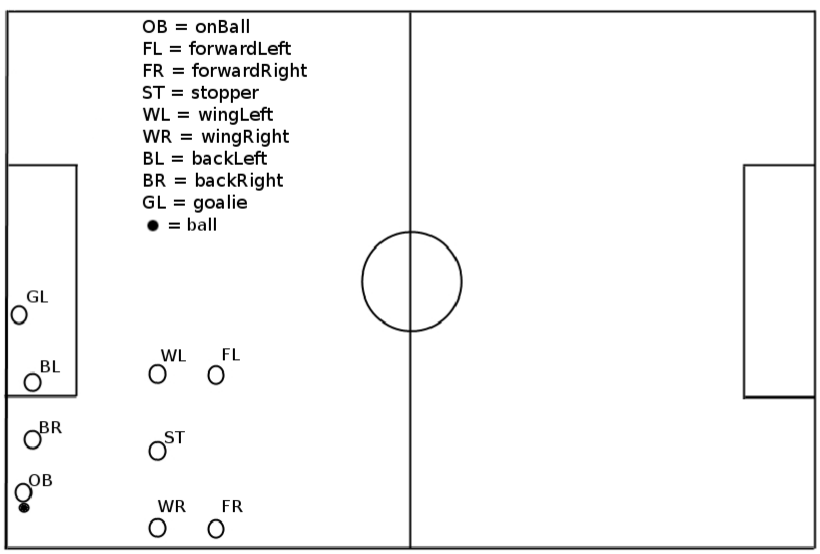

Figure 8: Shift Near Our Endline

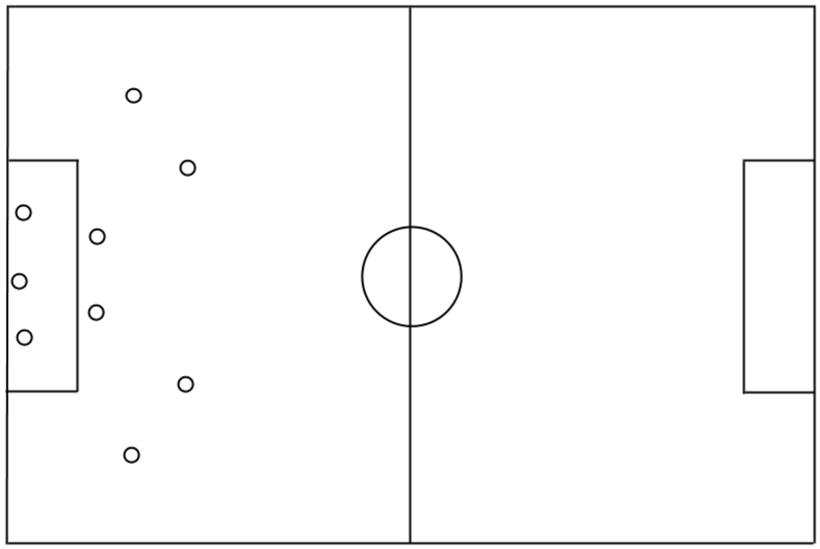

Figure 9: Defensive Goal Kick

11.2.2 Goal Kick

When goal kicks occur we try to position our team's agents in the most opportune places on the field. To achieve this we break from the normal formation and assume an entirely new one. Within these formations roles are meaningless, and therefore will not be referred to within this section.

During a defensive goal kick (i.e. goal kick for our team) we place three agents inside our goal box. The center agent (goalie) is .5 meters above the end line and centered with the goal. The other two agents are positioned around 2 meters diagonally away from either corner of the goal box. We have two agents .75 meters above the goal box and about a meter left and right of the center of the goal, and another two positioned 2.6 meters diagonally up-field and toward the sidelines from the corners of the goal box. We place the last two agents about 3 meters above either corner of the goal box. See Figure 9.

Once the agents reach their positions they pause, and will remain so until the RoboCup server beams the ball to one of the corners of the goal box and returns the play mode to normal play. We do not have a player kick the ball out of our goal box as we instead prefer to have the server move the ball to a less dangerous position to the side of the goal where we have an agent positioned in anticipation of the ball being moved.

During an offensive goal kick (i.e. goal kick for the opponent), we position two agents above the center of the opponents goal box, one at .75 meters and the other at about 3.5 meters. Two agents are positioned one meter inside and about 2.5 meters above either corner of the opponents goal box. We also place two agents 1.77 meters diagonally away from either corner of the goal box. The last three agents are positioned around our goal box to ensure we have some

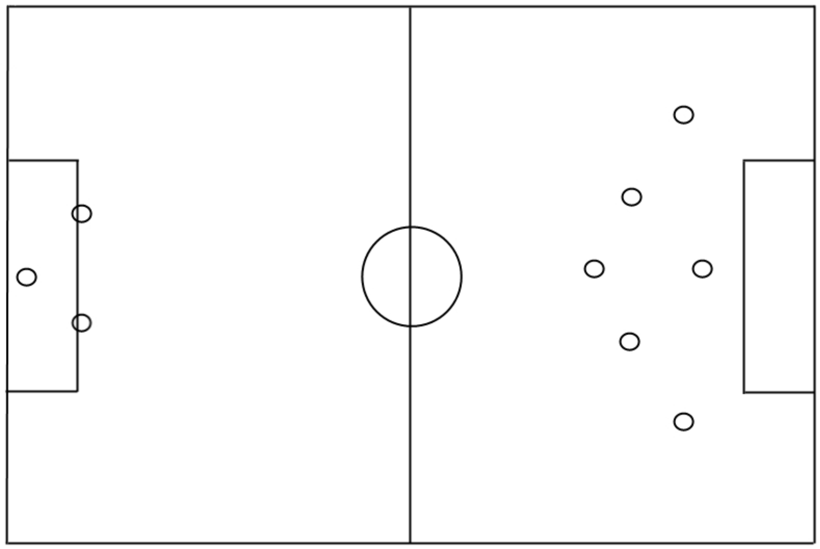

Figure 10: Offensive Goal Kick

defensive agents positioned near our goal. See Figure 10.

Similarly to the defensive goal kick, agents pause once they reach their positions and remain standing still until the ball gets beamed to the corner of the opponents goal box (where we have an agent waiting), or until one of the opposing team's agents brings the ball out of their goal box.

11.2.3 Corner Kick

When our team is given a corner kick we allow the agent, who is *onBall*, to intentionally kick the ball out of bounds over the opponents end line, giving the opponent a goal kick. While the *onBall* agent is executing this, all of the other agents are moving into the offensive goal kick formation (Section 11.2.2), leaving the closest position in the formation open for the *onBall* player to assume after the ball is kicked out of bounds. See Figure 11.

This strategy is used to try and catch the opponent out of position. Most teams move their agents over to defend against the corner kick, meaning that when we kick the ball out of bounds they are not in position to defend a goal kick leaving us one-on-one with their goalie.

11.2.4 Kick-Ins

When the ball gets kicked out of bounds the team who did not touch the ball last is awarded a kick-in. When this happens the RoboCup server beams any agent from the other team who moves too close to the ball. This becomes problematic for the *onBall* agent because (as mentioned in Section 11.1) this role is always positioned to where the ball is located. To account for this, we added logic to

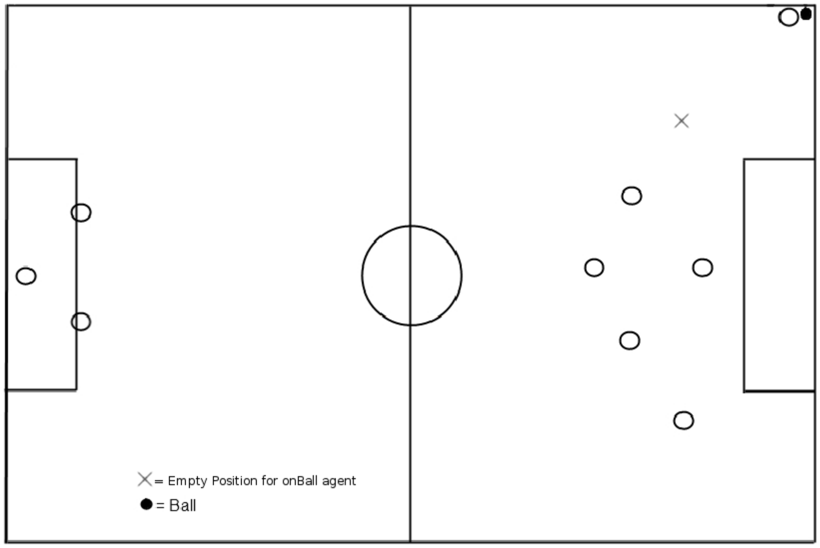

Figure 11: Corner Kick

recognize that a kick-in has been given to the opponent and therefore the agent should instead run the *collisionAvoidance* logic (discussed in Section 12.2) to avoid getting too close to the ball. To give room to the agent avoiding the ball, the other agents shift their positions half a meter away from the side line. This shift ensures that the *onBall* agent will not run into any of his teammates while waiting for the opposing team to move the ball back into play.

11.3 Assigning Agents to Roles

Given a desired team formation, we need to map players to roles (target positions on the field). Human soccer players specialize in different positions as they have different bodies and abilities, however, for us, the agents are all homogeneous, and so it is unnecessary to limit agents to constant specific roles. A naïve mapping having each player permanently mapped to one of the roles performs poorly due to the dynamic nature of the game. With such static roles an agent assigned to a defensive role may end up out of position and, without being able to switch roles with a teammate in a better position to defend, allow for the opponent to have a clear path to the goal. In this section, we present a dynamic role assignment algorithm. A role assignment algorithm can be thought of as implementing a role assignment *function*, which takes as input the state of the world, and outputs a one-to-one mapping of players to roles. We start by defining three properties that a role assignment function must satisfy (Section 11.3.1). We then construct a role assignment function that satisfies these properties (Section 11.3.2). Finally, we present a dynamic programming algorithm implementing this function (Section 11.3.3).

11.3.1 Desired Properties of a Valid Role Assignment Function

Before listing desired properties of a role assignment function we make a couple of assumptions. The first of these is that no two agents and no two role positions occupy the same position on the field. Secondly we assume that all agents move toward fixed role positions along a straight line at the same constant speed. While this assumption is not always completely accurate, the omnidirectional walk described in Section 10 gives a fair approximation of constant speed movement along a straight line.

We call a role assignment function *valid* if it satisfies the following three properties:

1. *Minimizing longest distance* - it minimizes the maximum distance from a player to target, with respect to all possible mappings.

2. *Avoiding collisions* - agents do not collide with each other as they move to their assigned positions.

3. *Dynamically consistent* - a role assignment function f is dynamically consistent if, given a *fixed* set of target positions, if f outputs a mapping m of players to targets at time T, and the players are moving toward these targets, f would output m for every time $t > T$.

The first two properties are related to the output of the role assignment function, namely the mapping between players and positions. We would like such a mapping to minimize the time until all players have reached their target positions because quickly doing so is important for strategy execution. As we assume all players move at the same speed, we start by requiring a mapping to minimize the maximum distance any player needs to travel. However, paths to positions might cross each other, therefore we additionally require a mapping to guarantee that when following it, there are no collisions. The third property guarantees that once a role assignment function f outputs a mapping, f is committed to it as long as there is no change in the target positions. This guarantee is necessary as otherwise agents might unduly thrash between roles thus impeding progress. In the following section we construct a valid role assignment function.

11.3.2 Constructing a Valid Role Assignment Function

Let M be the set of all one-to-one mappings between players and roles. If the number of players is n, then there are $n!$ possible such mappings. Given a state of the world, specifically n player positions and n target positions, let the *cost* of a mapping m be the n-tuple of distances from each player to its target, sorted in decreasing order. We can then sort all the $n!$ possible mappings based on their costs, where comparing two costs is done lexicographically. Sorted costs of mappings from agents to role positions for a small example are shown in Figure 12.

Denote the role assignment function that always outputs the mapping with the lexicographically smallest cost as f_v. Here we provide an informal proof sketch that f_v is a valid role assignment; we provide a longer, more thorough derivation in Appendix A.

Theorem 1. *f_v is a valid role assignment function.*

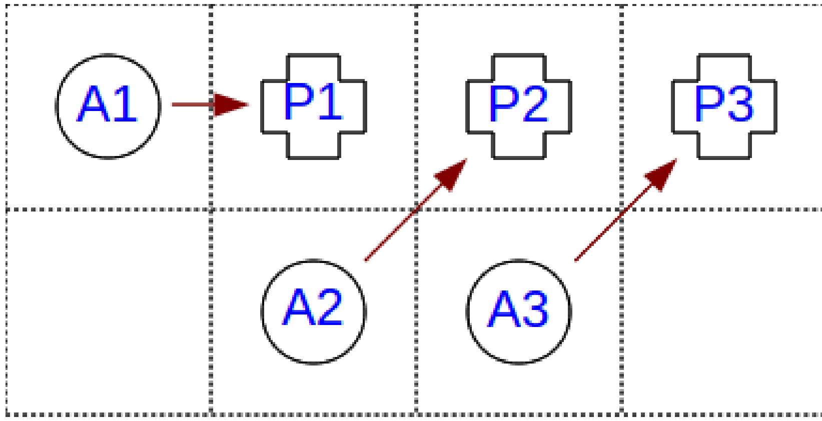

Figure 12: Lowest lexicographical cost (shown with arrows) to highest cost ordering of mappings from agents (A1,A2,A3) to role positions (P1,P2,P3). Each row represents the cost of a single mapping.

1:	$\sqrt{2}$ (A2→P2),	$\sqrt{2}$ (A3→P3),	1 (A1→P1)
2:	2 (A1→P2),	$\sqrt{2}$ (A3→P3),	1 (A2→P1)
3:	$\sqrt{5}$ (A2→P3),	1 (A1→P1),	1 (A3→P2)
4:	$\sqrt{5}$ (A2→P3),	2 (A1→P2),	$\sqrt{2}$ (A3→P1)
5:	3 (A1→P3),	1 (A2→P1),	1 (A3→P2)
6:	3 (A1→P3),	$\sqrt{2}$ (A2→P2),	$\sqrt{2}$ (A3→P1)

It is trivial to see that f_v minimizes the longest distance traveled by any agent (Property 1) as the lexicographical ordering of distance tuples sorted in descending order ensures this. If two agents in a mapping are to collide (Property 2) it can be shown, through the triangle inequality, that f_v will find a lower cost mapping as switching the two agents' targets reduces the maximum distance either must travel. Finally, as we assume all agents move toward their targets at the same constant rate, the distance between any agent and target will not decrease any faster than the distance between an agent and the target it is assigned to. This observation serves to preserve the lowest cost lexicographical ordering of the chosen mapping by f_v across all timesteps thereby providing dynamic consistency (Property 3).

The next section presents an algorithm that implements f_v.

11.3.3 Dynamic Programming Algorithm for Role Assignment

In UT Austin Villa's basic formation, presented in Section 11.1, there are nine different roles for each of the nine agents on the field. The goalie always fills the *goalie* role and the *onBall* role is assigned to the player closest to the ball. The other seven roles must be mapped to the agents by f_v. Additionally, when the goalie is closest to the ball, the goalie takes on both the *goalie* and *onBall* roles causing us to create an extra *goalieReplacement* role positioned right in front of the team's goal. When this occurs the size of the mapping increases to eight agents mapped to eight roles. As the total number of mapping permutations is

$n!$, this creates the possibility of needing to evaluate 8! different mappings.

Clearly f_v could be implemented using a brute force method to compare all possible mappings. This implementation would require creating up to $8! = 40,320$ mappings, then computing the cost of each of the mappings, and finally sorting them lexicographically and choosing the smallest one. However, as our agent acts in real time, and f_v needs to be computed during a decision cycle (0.02 seconds), a brute force method is too computationally expensive. Therefore, we present a dynamic programming implementation shown in Algorithm 1 that is able to compute f_v within the time constraints imposed by the decision cycle's length.

Algorithm 1 Dynamic programming implementation

```
1:  HashMap bestRoleMap = ∅
2:  Agents = {a_1, ..., a_n}
3:  Positions = {p_1, ..., p_n}
4:  for k = 1 to n do
5:      for each a in Agents do
6:          S = C(n-1, k-1) sets of k − 1 agents from Agents − {a}
7:          for each s in S do
8:              Mapping m_0 = bestRoleMap[s]
9:              Mapping m = (a → p_k) ∪ m_o
10:             bestRoleMap[a ∪ s] = mincost(m, bestRoleMap[a ∪ s])
11: return bestRoleMap[Agents]
```

Theorem 2. *Let A and P be sets of n agents and positions respectively. Denote the mapping $m := f_v(A, P)$. Let m_0 be a subset of m that maps a subset of agents $A_0 \subset A$ to a subset of positions $P_0 \subset P$. Then m_0 is also the mapping returned by $f_v(A_0, P_0)$.*

A key recursive property of f_v that allows us to exploit dynamic programming is expressed in Theorem 2. This property stems from the fact that if within any subset of a mapping a lower cost mapping is found, then the cost of the complete mapping can be reduced by augmenting the complete mapping with that of the subset's lower cost mapping. The savings from using dynamic programming comes from only evaluating mappings whose subset mappings are returned by f_v. This is accomplished in Algorithm 1 by iteratively building up optimal mappings for position sets from $\{p_1\}$ to $\{p_1, ..., p_n\}$, and using optimal mappings of $k - 1$ agents to positions $\{p_1, ..., p_{k-1}\}$ (line 8) as a base when constructing each new mapping of k agents to positions $\{p_1, ..., p_k\}$ (line 9), before saving the lowest cost mapping for the current set of k agents to positions $\{p_1, ..., p_k\}$ (line 10).

An example of the mapping combinations evaluated in finding the optimal mapping for three agents through the dynamic programming approach of Algorithm 1 can be seen in Table 4. In this example we begin by computing the distance of each agent to our first role position. Next we compute the cost of all possible mappings of agents to both the first and second role positions and save off the lowest cost mapping of every pair of agents to the the first two positions. We then proceed by sequentially assigning every agent to the third position and compute the lowest cost mapping of all agents mapped to all three positions. As

{P1}	{P2,P1}	{P3,P2,P1}
A1→P1	A1→P2, f_v(A2→P1)	A1→P3, f_v({A2,A3}→{P1,P2})
A2→P1	A1→P2, f_v(A3→P1)	A2→P3, f_v({A1,A3}→{P1,P2})
A3→P1	A2→P2, f_v(A1→P1)	A3→P3, f_v({A1,A2}→{P1,P2})
	A2→P2, f_v(A3→P1)	
	A3→P2, f_v(A1→P1)	
	A3→P2, f_v(A2→P1)	

Table 4: All mappings evaluated during dynamic programming using Algorithm 1 when computing an optimal mapping of agents A1, A2, and A3 to positions P1, P2, and P3. Each column contains the mappings evaluated for the set of positions listed at the top of the column.

all subsets of an optimal (lowest cost) mapping will themselves be optimal, we need only evaluate mappings to all three positions which include the previously calculated optimal mapping agent combinations for the first two positions.

Recall that during the *kth* iteration of the dynamic programming process to find a mapping for n agents, where k is the current number of positions that agents are being mapped to, each agent is sequentially assigned to the *kth* position and then all possible subsets of the other $n-1$ agents are assigned to positions 1 to $k-1$ based on computed optimal mappings to the first $k-1$ positions from the previous iteration of the algorithm. These assignments result in a total of $\binom{n-1}{k-1}$ agent subset mapping combinations to be evaluated for mappings of each agent assigned to the *kth* position. The total number of mappings computed for each of the n agents across all n iterations of dynamic programming is thus equivalent to the sum of the $n-1$ binomial coefficients. That is,

$$\sum_{k=1}^{n}\binom{n-1}{k-1} = \sum_{k=0}^{n-1}\binom{n-1}{k} = 2^{n-1}$$

Therefore the total number of mappings that must be evaluated using our dynamic programming approach is $n2^{n-1}$. For $n=8$ we thus only have to evaluate 1024 mappings which is very manageable. For future competitions it is projected that teams will increase to 11 agents to match that of actual soccer. In this case, where $n=10$, the number of mappings to evaluate will only increase to 5120 which is drastically less than the brute force method of evaluating all possible $10! = 3,628,800$ mappings.

11.4 Voting Coordination System

In order for agents on a team to assume correct positions on the field they all must coordinate and agree on which mapping of agents to roles to use. If every agent had perfect information of the locations of the ball and its teammates this would not be a problem as each could independently calculate the optimal mapping to use. Agents do not have perfect information, however, and are limited to noisy measurements of the distance and angle to objects within a restricted vision cone (120°). Fortunately agents can share information with each other every other simulation cycle (40 ms). The bandwidth of this communication channel is very limited, however, as only one agent may send a message at time and messages are limited to 20 bytes.

We utilize the agents' limited communication bandwidth in order to coordinate role mappings, as follows. Each agent is given a rotating time slice to communicate information which is based on the uniform number of an agent. When it is an agent's turn to send a message it broadcasts to its teammates its current position, the position of the ball, and also what it believes the optimal mapping should be. By sending its own position and the position of the ball, the agent provides necessary information for computing the optimal mapping to those of its teammates for which these objects are outside of their view cones. Sharing the optimal mapping of agents to role positions enables synchronization between the agents, as follows.

First note that just using the last mapping received is dangerous, as it is possible for an agent to report inconsistent mappings due to its noisy view of the world. This can easily occur when an agent falls over and accumulates error in its own localization. Additionally, messages from the server are occasionally dropped or received at different times by the agents preventing accurate synchronization. To help account for inconsistent information, a sliding window of received mappings from the last n time-slots is kept by each agent where n is the total number of agents on a team. Each of these kept messages represents a single vote by each of the agents as to which mapping to use. The mapping chosen is the one with the most votes or, in the case of a tie, the mapping tied for the most votes with the most recent vote cast for it. By using a voting system, the agents on a team are able to synchronize the mapping of agents to role positions in the presence of occasional dropped messages or an agent reporting erroneous data. Although this voting system is fairly simplistic, and we make no guarantees as to its ability to keep the agents synchronized, in practice it works very well.

11.5 Formation Evaluation

To test how our formation and role positioning system[5] affects the team's performance we created a number of teams to play against by modifying the positioning system of UT Austin Villa that was used in the competition.

AllBall No formations and every agent except for the goalie just goes to the ball.

Static Each role is statically assigned to an agent based on its uniform number.

Defense Defensive formation in which only two agents are in the offensive group (one on the ball and the other directly behind the ball)

Boxes Field is divided into fixed boxes and each agent is dynamically assigned to a home position in one of the boxes. Similar to the positioning system used in [11].

Results of UT Austin Villa playing against these modified versions of itself are shown in Table 5. We see that a very defensive formation used by the *Defense* agent hurts performance a little likely because the best defense is a good offense.

[5] Video demonstrating our positioning system can be found online at http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/AustinVilla3DSimulationFiles/2011/html/positioning.html

Table 5: Full game results, averaged over 100 games. Each row corresponds to an agent with varying formation and positioning systems as described in Section 11.5. Entries show the goal difference from 10 minute games versus our agent using the dynamic role positioning system and formation described in Section 11. Values in parentheses are the standard error.

Team	Goal Difference
Defense	.29 (.06)
Static	.32 (.07)
AllBall	.43 (.09)
Boxes	1.26 (.10)

Dynamically assigning roles is better than statically fixing them as is clear in the degradation in performance of the *Static* agent. Having and maintaining formations is also important which is evident by the positive goal difference recorded when playing against the *AllBall* agent. The poor performance of the *Boxes* agent, in which the positions on the field are somewhat static and not calculated as relative offsets to the ball, underscores the importance of being around the ball and adjusting positions on the field based on the current state of the game.

12 General Locomotion in the Field

Having specified how the agent walks in Section 10, and the team's overall formation in Section 11, this section covers additional details regarding how agents behave on the field. Section 12.1 explains how which agent should go to the ball is determined. A system used to avoid collisions is given in Section 12.2. Movement and actions around the ball including facing the ball (Section 12.3), how the ball is approached (Section 12.4), where to move the ball (Section 12.5), how to dribble (Section 12.6), and when to kick (Section 12.7) follow.

12.1 Closest to Ball Heuristic

In Section 11.1 we mention that the closest player to the ball is assigned the *onBall* role and is instructed to go to the ball. In order to measure "closeness" we do not purely use Euclidean distance, however, as certain positions such as being behind the ball instead of in front of it are more advantageous. An agent is considered to be in front of the ball if its X coordinate is greater than that of the ball's X coordinate. If an agent is in front of the ball it will typically have to take time to circle and walk around behind the ball in order to dribble the ball forward toward the opponent's goal. For this reason we add 1 meter to the distances of agents in front of the ball when determining the agent closest to the ball.

When the ball is close to either end of the field we modify the definition of being in front of the ball to take into consideration that agents near the opponent's goal want to move the ball toward the center of the field (toward the opponent's goal) and agents near their own goal want to push the ball out to the sides (away from their goal). For this reason whenever the ball is to either side of the goal (its Y coordinate is outside the closest goal post's Y coordinate), and the ball's distance to the nearest endline is less than the distance between a goal post and the closest corner of the field to the goal post (approximately 6 meters), we declare an agent to be in front of the ball if on offense the agent is closer than the ball to the goal post nearest the ball, or on defense if the agent is farther than the ball from the goal post nearest to the ball.

Another situation in which we adjust the measure of the distance an agent is considered to be from the ball is when an agent has fallen. As it takes time after a fall for an agent to get back up, we add an extra 1.5 meters to the distance a fallen agent is considered to be from the ball. The only time we do not add in this extra distance measure for a fallen agent is when the agent has fallen very near (within .65 meters) of the ball. In this case, when the fallen agent is almost on top of the ball, having another agent assume the *onBall* role will likely force that agent to have to navigate around and possibly trip over the fallen agent when moving toward the ball.

One last factor we take into consideration when computing our "closeness" to the ball measure is that the goalie, as discussed in Section 9.1, should never leave our goal box. We therefore declare the goalie to be a very large distance from the ball unless the ball has entered our goal box. Below is pseudocode for how we calculate distances when determining which agent is closest to the ball.

```
// Function for computing the adjusted distance (in meters)
// an agent is to the ball.
function getClosenessToBallMeasure(agent) {
  // Goalie should not be closest if ball is in own goal box
  if agentIsGoalie and !ballInOwnGoalBox
    return 1000;

  // Adjustment value to add to distance agent is from ball
  adjust = 0.0;

  // Agent has fallen but not right on top of ball
  if agentIsFallen and agentDistToBall > .65
    adjust += 1.5; // Added distance for having fallen

  // Ball is to the sides of the goals
  if abs(ball_Y) > HALF_GOAL_Y {
    // Ball close to own goal
    if ball_X < -HALF_FIELD_X + (HALF_FIELD_Y-HALF_GOAL_Y) {
      if ball_Y > 0
        nearestPost = Position(-HALF_FIELD_X, HALF_GOAL_Y);
      else
        nearestPost = Position(-HALF_FIELD_X, -HALF_GOAL_Y);

      // Agent is in front of ball
      if agentDistToNearestPost > ballDistToNearestPost
        adjust += 1.0; // Added distance to walk around ball
    }
    // Ball close to opponent's goal
    else if ball_X > HALF_FIELD_X - (HALF_FIELD_Y-HALF_GOAL_Y) {
```

```
      if ball_Y > 0
        nearestPost = Position(HALF_FIELD_X, HALF_GOAL_Y);
      else
        nearestPost = Position(HALF_FIELD_X, -HALF_GOAL_Y);

      // Agent is in front of ball
      if agentDistToNearestPost < ballDistToNearestPost
        adjust += 1.0; // Added distance to walk around ball
    }
  }
  // Agent is in front of ball
  else if agent_X >= ball_X
    adjust += 1.0; // Added distance to walk around ball

  return agentDistToBall + adjust;
}
```

12.2 Collision Avoidance

Although the positioning system discussed in Section 11 is designed to avoid assigning agents to positions that might cause them to collide, external factors outside of the system's control, such as falls and the movement of the opposing team's agents, still result in occasional collisions. To minimize the potential for these collisions the agents employ an active collision avoidance system. When an obstacle, such as a teammate, is detected in an agent's path the agent will attempt to adjust its path to its target in order to maneuver around the obstacle. This adjustment is accomplished by defining two thresholds around obstacles: a *proximity* threshold at 1.25 meters and a *collision* threshold at .5 meters from an obstacle. If an agent enters the *proximity* threshold of an obstacle it will adjust its course to be tangent to the obstacle thereby choosing to circle around to the right or left of said obstacle depending on which direction will move the agent closer to its desired target. Should the agent get so close as to enter the *collision* proximity of an obstacle it must take decisive action to prevent an otherwise imminent collision from occurring. In this case the agent combines the corrective movement brought about by being in the *proximity* threshold with an additional movement vector directly away from the obstacle. Figure 13 illustrates the adjusted movement of an agent when attempting to avoid a collision with an obstacle.

12.3 Ball Facing

When an agent is assigned to move to a new role position on the field, as described in Section 11, the agent both turns toward and moves to the new target position as described in Section 10.2. Once the agent gets within .5 meters of its target role position it no longer attempts to face in the direction of its target position, however, and instead turns to face the ball as it finishes moving toward its target. This is done so that slight adjustments to an agent's target, brought about by small movement or noise in the position of the ball, do not result in sudden quick turns in place that might destabilize the agent as it adjusts its position to the revised target. Facing the ball when stopped also

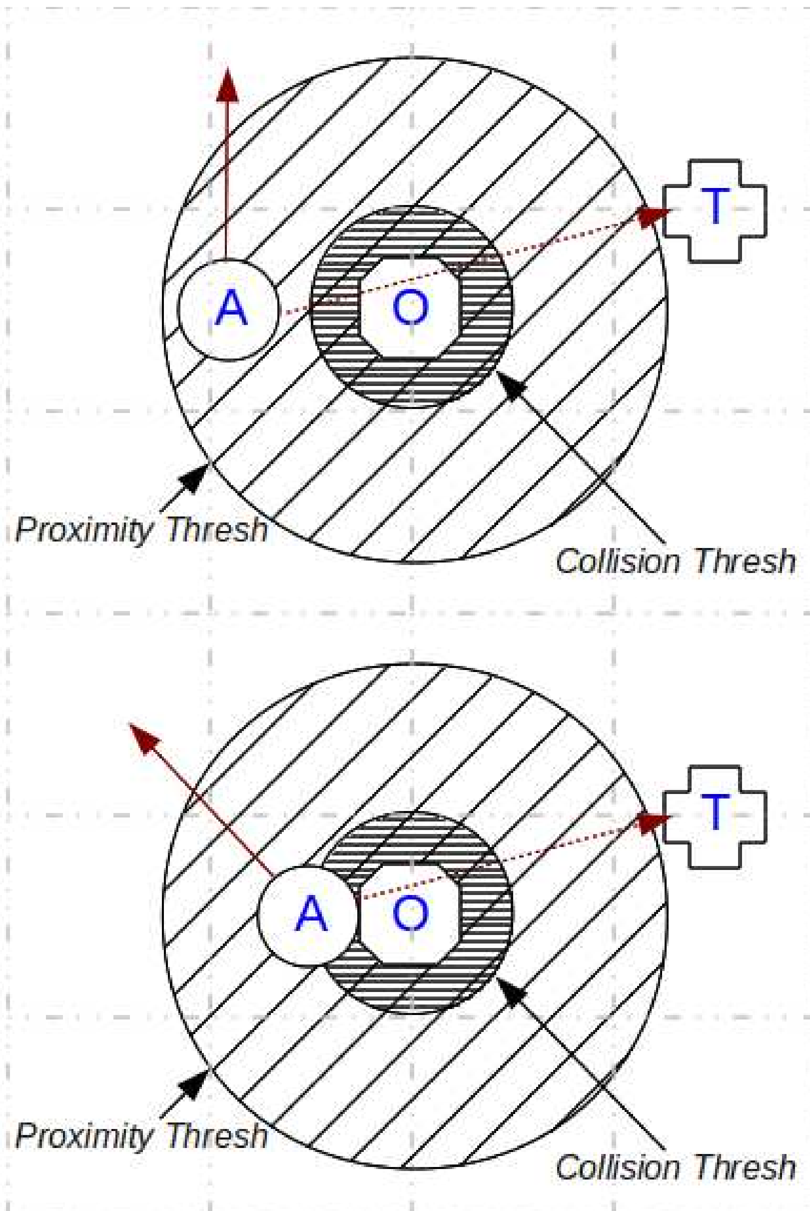

Figure 13: Collision avoidance examples where agent A is traveling to target T but wants to avoid colliding with obstacle O. The top diagram shows how the agent's path is adjusted if it enters the *proximity* threshold of the obstacle while the bottom diagram depicts the agent's movement when entering the *collision* threshold. The dotted arrow is the agent's desired path while the solid arrow is the agent's corrected path to avoid a collision.

allows for agents to quickly move to the ball, without needing to turn, should they become the closest agent to the ball as determined in Section 12.1.

12.4 Ball Approach

When the agent approaches the ball to dribble or kick the ball it moves toward a target position a little behind the ball that is in line with the direction the agent wants to move the ball. Should the agent be in front of the ball, meaning that if it were to walk straight to its desired target behind the ball it would end up walking through the ball, the agent instead picks a target to move to that is .5 meters either left or right from the ball along a line perpendicular to the direction from the agent to the ball. This provides a waypoint for the agent to move through, along an efficient path around the ball, as opposed to directly walking up to the ball and then having to walk all the way around it.

If an opponent agent is within a meter of the ball, and the ball is between the opponent agent and our goal, it is likely that the opposing agent is going to move the ball toward our goal. If our agent were to walk straight toward the ball, and the opponent agent does start dribbling, chances are that the opponent agent will move the ball past our agent and need to be chased after. Our agent recognizes this situation when going to the ball and adjusts its target position to be further behind the ball, and along the anticipated path that the opponent agent is projected to dribble the ball, so as to be position to intercept the ball should the opponent agent move it. This can be thought of as approaching the ball with a good angle for pursuit.

12.5 Reflex-based Strategy for Navigation with the Ball

By default our agents attempt to drive (move) the ball toward the opponent's goal with the target destination being the center of the opponent's goal. However, in many cases, the fastest way to drive a ball to this target point is different than just dribbling/kicking it directly to the target. For instance, when the agent is approximately aligned behind the ball facing the target direction, frequently it is faster to start dribbling the ball and slightly adjust the path of the ball while dribbling, then trying to align exactly in the target direction before starting to dribble. Other times when it is better to start dribbling the ball immediately, instead of waiting to align in the target direction, include when an opponent is close by, and there is no time to turn to face the exact target direction, and when an opponent is blocking the path to the target. This section describes a simple, reflex-based navigation strategy used when driving the ball, which performed robustly during the RoboCup 2010 and RoboCup 2011, competitions.

The general idea behind this strategy is that given a desired target direction for the ball to move in, and given the agent's current direction facing the ball, a decision is made as to whether the agent should just move the ball in the direction of its current heading based on the current state of the game. This strategy is encapsulated in a function named *shouldMoveBallInCurrentDirection()* which returns `true` when the agent should move the ball forward along its current direction relative to the ball. This function is roughly implemented as follows:

```
function shouldMoveBallInCurrentDirection(agentDirection,
                                           desiredDirection, ...) {
  if ballWouldGoInsideGoal
    return true;
  else if ballGoingOutsideFieldBounds
    return false;
  else if agentDirection is too backwards
    return false; // allow to dribble only mildly backwards
  else if opponentsAreFar
    return false; // we have time to better align
  else if opponentGetsClose
          and opponentDoesNotBlockAgentPath
          and goingForwardGetsBallCloserToOpponentGoal
    return true;
  else if opponentIsNearBall
    return true; // do not let opponent reach the ball
  else
    return false; // on all other cases, try to align better
}
```

Using this method, the agent was able to quickly navigate between obstacles, without the need for a complex path planning algorithm. Note that at each moment, the agent ignores all but the closest obstacle, making this a reflex-based strategy rather than a planning with lookahead strategy.

12.6 Dribbling

Dribbling the ball amounts to walking through the center of the ball in the desired direction that the agent wants to move the ball. When the agent is close to the ball, and is attempting to position itself behind the ball, it always faces the ball, as mentioned in Section 10.2, so that it can quickly walk forward and move the ball should an opposing agent approach. When circling the ball to dribble the agent uses collision avoidance (Section 12.2) with a *proximity* threshold of .5 meters and a *collision* threshold of .35 meters to avoid running into the ball.

12.7 When to Kick

When our agent is on the ball it has the option of kicking the ball (discussed in Section 13) or dribbling the ball in its target direction. By default the agent chooses to dribble the ball as doing so is more reliable than kicking and helps to maintain possession of the ball. However if an opponent agent is less than 3.5 meters from the ball, and within 45° of our desired heading, the agent attempts to kick the ball so as to be able to move the ball past the opponent without risking colliding with the opponent while dribbling. If an opponent gets within a meter of the ball the agent aborts kicking and reverts back to dribbling as there might not be enough time to kick the ball before the opponent reaches it. Also, if the agent spends more than 15 seconds trying to line up a kick, it will give up on the kick and just dribble the ball.

13 Kicking

To motivate some of the design decisions in our kick engine which we discuss in depth later in this section, we first present the desired qualities of the engine. For a kick to be broadly applicable, it needs to be agile, robust, versatile, and easily and concisely parameterizable. *Agility* refers to taking shots quickly. *Robustness* entails taking accurate and powerful shots in spite of positioning errors (e.g., without the agent being perfectly lined up with the ball). *Versatility* refers to being able to kick in multiple directions from multiple ball starting locations. The parameterization criterion serves to facilitate learning optimized kicks.

13.1 Kick Engine Implementation

To achieve these criteria, our kick engine employs a system of defining and dynamically computing smooth curves which guide the foot's trajectory through the ball at high speed and in the desired direction. We use Cubic Hermite Splines to define the foot trajectories. Agility and robustness are achieved by defining the kick trajectory relative to the ball in Cartesian space. Unlike our previous year's team which used fixed joint angle skills exclusively, the current agents do not have to tip-toe e.g., directly behind the ball at a set distance in order to kick the ball e.g., forward. Instead, the kick engine dynamically computes the trajectory of the foot once the agent is close enough to the ball, regardless of whether the agent finished positioning or whether the agent was able to position itself precisely relative to the ball. Versatility is achieved because multiple directional kicks can be defined and used at will. Learning and optimization of kicks is facilitated by the parameterization of the foot trajectories in terms of a sparse set of control (way-) points. The flow of the kick engine follows.

First, a kick is selected, and the agent approaches the ball (Section 13.1.1). Once close enough to the ball, it shifts its weight onto the support foot and computes the kicking foot trajectory necessary to perform the desired kick (Section 13.1.2). At each time step during the kick, the kick engine interpolates the control (way-) points defined in the kick skill file (Section 13.1.5) to produce a target pose for the foot in Cartesian space (Section 13.1.3). Finally, an IK solver computes the necessary joint angles of the kicking leg, and these angles are fed to the joint PID controllers (Section 13.1.4). Figure 14 illustrates the program flow of the kick engine.

13.1.1 Kick Choice and Ball Approach

As the agent approaches the ball, it must decide which type of kick to attempt (Section 13.1.6 describes the options) and whether to use the left or right foot. Each kick skill definition includes a target offset of the agent relative to the ball. Choosing a kick reduces to choosing the target with the lowest cost for the agent to move to. We calculate the cost of each target through the following

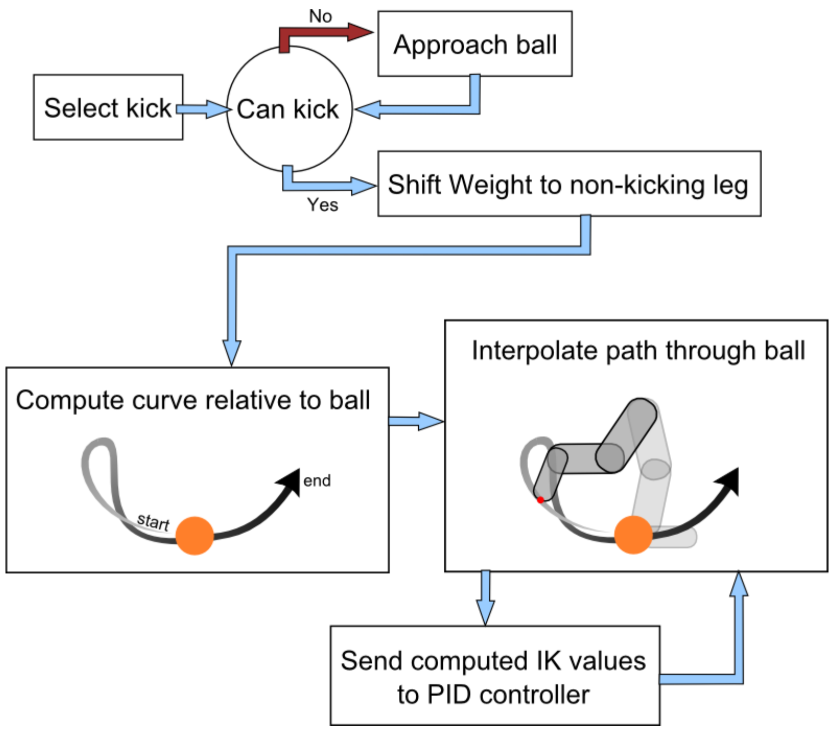

Figure 14: The flow of the agent deciding when to kick the ball and how to interpolate the curve created relative to the ball.

variables and formula:

$$\begin{aligned}
distCost &= \left|\text{agentPosition} - \text{targetOffsetPosition}\right| / m \\
turnCost &= \frac{\left|\text{agentOrientation} - \text{targetOrientation}\right|}{360^\circ} \\
ballPenalty &= \begin{cases} .5 & \text{if ball is in path to target offset} \\ 0 & \text{otherwise} \end{cases} \\
kickCost &= \text{distCost} + \text{turnCost} + \text{ballPenalty}
\end{aligned}$$

The chosen target is approached using the walk engine. During approach, the kick engine continuously checks if the agent is close enough to kick by using the IK solver to determine if the foot can reach most ($> 90\%$) of the points along the trajectory for the chosen kick.

13.1.2 Dynamically Compute Kick Trajectory

Once the agent has shifted its weight in preparation for a kick, it notes the ball's position with respect to itself (specifically its torso, the root of the leg kinematic chains). This offset is added to the control points in the kick skill file to dynamically compute the exact curve of the foot with respect to the agent's torso.

13.1.3 Interpolate Kick Trajectory

The control points defined in the kick skill files are used to compute a smooth 3D curve. We use the Cubic Hermite Spline formulation to interpolate the control points because Hermite Splines yield curves with C^1 continuity which pass through all control points [4]. The time offset from the start of the kick is normalized to the range $[0-1]$ (0 is the start of the kick; 1 is the end), and the normalized offset is used to sample the Hermite Spline. The kick skill files also define the Euler angles (roll, pitch, and yaw) of the foot at each control point. These angles are linearly interpolated.

13.1.4 Kick Inverse Kinematics

For the inverse kinematics calculations, we used OpenRAVE's [5] analytic inverse kinematics solver. The OpenRAVE IK solver can process arbitrary forward kinematic chains defined in XML and produce fast C++ source code that solves the inverse kinematics. Note that the time-consuming analytic processing is done offline, and the fast C++ code can be queried hundreds of times at each time step without a significant computational cost.

13.1.5 Kick Skill Definition

Extending the skill definition files presented in Section 7 to allow Cartesian coordinate plus Euler angle waypoints for each foot, we predefine all six degree of freedom positions of the foot for a given curve at any linear time through the curve.

13.1.6 Directional Kicks

We defined five kicks that assume that the ball is in front of the agent such that it can kick directly forward and at 45° and 90° angles either outward or inward, depending on which leg is used. We also created directional kicks which assume that the ball is to the side of or behind one of the legs. See Figure 15.

13.2 Kick Optimization

We can then optimize the waypoints (three to five per kick) for kicked distance and speed through CMA-ES. This then allows us to have multiple directional kicks[6] defined through simple curves as we do not have to dedicate large amounts of time tweaking each one and can create rough paths to guide the initial seed of the agent's kick.

In order to learn the parameters for a kick we set up an optimization task where the agent approaches the ball from ten different angles along a half circle arc around the ball and attempts to kick the ball toward a specific target. The parameters being optimized are the XYZ and RPY values of the waypoints that define the curve of the kick, how quickly the kicking foot moves through the curve, and also the target offset from the ball to move toward during the kick approach. The fitness of an agent is measured by the average distance the

[6] Videos of the kicks and the optimization process can be found online at http://www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/AustinVilla3DSimulationFiles/2011/html/kick.html

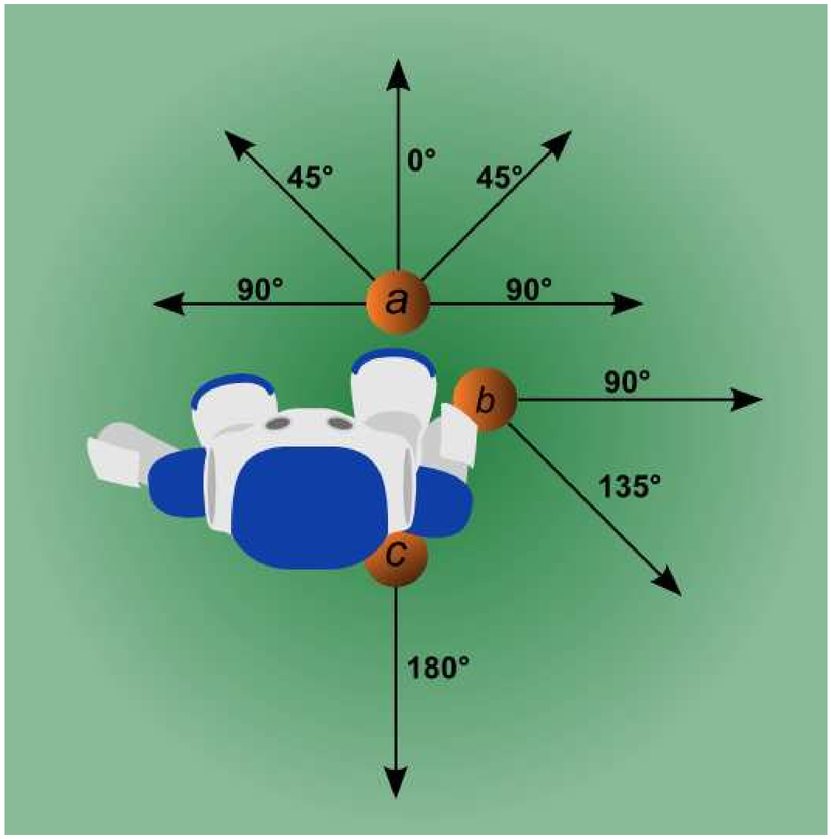

Figure 15: The agent can dynamically kick the ball in varied directions with respect to the placement of the ball at *a, b,* and *c*.

ball travels toward the target across all kick attempts. The agent is given a penalty fitness of -1 for every kick during which it falls over, runs into the ball, or is not able to kick the ball after ten seconds have passed. Penalizing the agent for taking too long to kick encourages kicking agility while having the agent approach the ball from multiple angles and penalizing for falling promote kicking robustness.

13.3 Kick Performance

While our kicking system shows a lot of promise, we found out after the competition that our agent does slightly better without kicking turned on during self play. A version of our agent with the kicking system turned off was able to beat our agent that does kick by an average of .15 goals per game across 100 games with a standard error of .07. This resulted in a tally of 27 wins for the agent that does not kick, 12 wins for that agent that does kick, and 61 ties. We believe the reason for this slight degradation in performance when kicking is due to our kicking agent needing to slow down a little when approaching the ball to kick it, instead of maintaining a full speed walk while dribbling the ball, so as to not accidentally run into the ball. Additionally we have yet to implement a strategy for passing and only kick in the direction we want to dribble if an opponent agent is approaching to take the ball away. We therefore include a description of the kick in this paper as a key component of the overall agent, even though it was not necessary for winning this year's competition.

With better tuning such that the agent can approach the ball without needing to slow down, and the addition of a strategy to take full advantage of the ability for kicking to quickly move the ball, we expect our kick system to provide a substantial gain in the performance of the agent. The kicking system has already shown some promise when used with walks that are not as effective at dribbling as our current walk. When playing kicking and non-kicking versions of our agent with slow *initial* walk parameters, as described in Section 10.3, against each other the kicking agent scored 8 goals while the non-kicking agent failed to score.

14 Penalty kicks

At the 2011 RoboCup competitions, "penalty kicks" were used for tie-breaking in the knock-out stages. As in real soccer, the only agents involved in a penalty kick are a striker from the offense team and the goalie from the defense team. However, unlike in real soccer, where the striker takes a kick from within the penalty area, in the 3D simulation competition, the ball is placed at the center of the field. The offense team is given 40 seconds to try and score a goal. Thus, the striker may plan a sequence of steps, such as dribbling the ball along a certain trajectory, before "getting past" the goalie and scoring a goal. During a penalty kick the goalie is not allowed to leave the goal box. Although UT Austin Villa was never faced with a penalty kick situation in the 2011 competitions, below we briefly describe the goalie and striker agents we had designed for the penalty kick scenario.

When the ball is outside the goal box, our goalie assumes its normal behavior as described in Section 9. An exception to this is if the ball, while still outside the goal box, gets within close range of the goal (3.5 meters). At this point the goalie walks up toward the top edge of the goal box to better cut off shooting angles in anticipation of a shot on goal. If the ball is within the goal box, the goalie walks directly toward the ball and dribbles it out of the goal box. Should the opposing team's striker kick the ball over the endline for a goal kick, the goalie purposely avoids kicking the ball to start play again so that time will run out on the penalty kick attempt.

Our striker agent dribbles the ball to a position roughly a meter outside the goal box. Once there the striker chooses to shoot either to the left or right side of the goal depending on the position of the goalie: the striker executes a kick action in a direction that bisects the largest angle between the opposing goalie and any one of the goal posts. If this kick is unsuccessful, the striker intercepts the ball and dribbles it toward the goal, as it would during normal course of play. If over 25 seconds have passed, and less than 15 seconds are left to try and score, the striker gives up on kicking and just tries to dribble the ball into the goal.

We have not systematically tested our goalie and striker agents either against each other or when paired with other teams' agents. Such testing is planned for

Table 6: Full game results, averaged over 100 games. Each row corresponds to an agent from the RoboCup 2011 competition, with its rank therein achieved. Entries show the goal difference from 10 minute games versus our agent. Values in parentheses are the standard error.

Rank	Team	Goal Difference
3	apollo3d	1.45 (0.11)
5-8	boldhearts	2.00 (0.11)
5-8	robocanes	2.40 (0.10)
2	cit3d	3.33 (0.12)
5-8	fcportugal3d	3.75 (0.11)
9-12	magmaoffenburg	4.77 (0.12)
9-12	oxblue	4.83 (0.10)
4	kylinsky	5.52 (0.14)
9-12	dreamwing3d	6.22 (0.13)
5-8	seuredsun	6.79 (0.13)
13-18	karachikoalas	6.79 (0.09)
9-12	beestanbul	7.12 (0.11)
13-18	nexus3d	7.35 (0.13)
13-18	hfutengine3d	7.37 (0.13)
13-18	futk3d	7.90 (0.10)
13-18	naoteamhumboldt	8.13 (0.12)
19-22	nomofc	10.14 (0.09)
13-18	kaveh/rail	10.25 (0.10)
19-22	bahia3d	11.01 (0.11)
19-22	l3msim	11.16 (0.11)
19-22	farzanegan	11.23 (0.12)

future work.

15 Competition Results

UT Austin Villa 2011 won all 24 of its games during the RoboCup 2011 3D simulation competition, scoring 136 goals and conceding none. Even so, competitions of this sort do not consist of enough games to validate that any team is better than another by a statistically significant margin. In order to validate the results of the competition, in Table 6 we show the performance of our team when playing 100 games against each of the other 21 teams' released binaries from the competition. UT Austin Villa won by at least an average goal difference of 1.45 against every team. Furthermore, of these 2100 games played to generate the data for Table 6, our agent won all but 21 of them which ended in ties (no losses). The few ties were all against three of the better teams: apollo3d, boldhearts, and robocanes. We can therefore conclude that UT Austin Villa was the rightful champion of the competition.

While there were multiple factors and components that contributed to the success of UT Austin Villa in winning the competition, its omnidirectional walk was the one which proved to be the most crucial. When switching out the omnidirectional walk developed for the 2011 competition with the fixed directional walk used in the 2010 competition, and described in [12], the team did not fare nearly as well. The agent with the previous year's walk had a negative average goal differential against nine of the teams from the 2011 competition, suggesting a probable tenth place finish. Also this agent lost to our 2011 agent by an average of 6.32 goals across 100 games with a standard error of .13

16 Summary and Discussion

We have presented the architecture, design decisions, and components of the UT Austin Villa 2011 RoboCup 3D simulation league team. These components include an omnidirectional walk engine and associated walk parameter optimization framework, an inverse kinematics based kicking architecture, and a dynamic role and formation positioning system.

Our ongoing research agenda includes applying what we have learned in simulation to the actual Nao robots which we use to compete in the Standard Platform league of RoboCup. For next year's competition we expect to better integrate and utilize our kicking system in order to improve the performance of the team. Additionally, we would like to learn and add further parameter sets to our team's walk engine for important subtasks such as goalie positioning to get ready to block a shot.

Acknowledgments

This work has taken place in the Learning Agents Research Group (LARG) at UT Austin. Thanks especially to UT Austin Villa 2011 team members Michael Quinlan, Nick Collins, and Art Richards. Also thanks to Yinon Bentor and Suyog Dutt Jain for contributions to early versions of the optimization framework employed by the team. LARG research is supported in part by NSF (IIS-0917122), ONR (N00014-09-1-0658), and the FHWA (DTFH61-07-H-00030). Patrick MacAlpine and Samuel Barrett are supported by NDSEG fellowships.

References

[1] Aldebaran Humanoid Robot Nao. `http://www.aldebaran-robotics.com/eng/`.

[2] Open Dynamics Engine. `http://www.ode.org/`.

[3] SimSpark. `http://simspark.sourceforge.net/`.

[4] E. Angel. *Interactive Computer Graphics*. Pearson Education, Inc., 5th edition, 2009.

[5] R. Diankov and J. Kuffner. Openrave: A planning architecture for autonomous robotics. Technical Report CMU-RI-TR-08-34, Robotics Institute, Pittsburgh, PA, July 2008.

[6] D. Fox, W. Burgard, F. Dellaert, and S. Thrun. Monte carlo localization: Efficient position estimation for mobile robots. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, pages 343–349, 1999.

[7] C. Graf, A. Härtl, T. Röfer, and T. Laue. A robust closed-loop gait for the standard platform league humanoid. In C. Zhou, E. Pagello, E. Menegatti, S. Behnke, and T. Röfer, editors, *Proceedings of the Fourth Workshop on Humanoid Soccer Robots in conjunction with the 2009 IEEE-RAS International Conference on Humanoid Robots*, pages 30 – 37, Paris, France, 2009.

[8] N. Hansen. *The CMA Evolution Strategy: A Tutorial*, January 2009. http://www.lri.fr/~hansen/cmatutorial.pdf.

[9] S. Kalyanakrishnan and P. Stone. Learning complementary multiagent behaviors: A case study. In *RoboCup 2009: Robot Soccer World Cup XIII*, pages 153–165. Springer, 2010.

[10] P. MacAlpine, D. Urieli, S. Barrett, S. Kalyanakrishnan, F. Barrera, A. Lopez-Mobilia, N. Ştiurcă, V. Vu, and P. Stone. UT Austin Villa 2011: A champion agent in the RoboCup 3D soccer simulation competition. In *Proc. of 11th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, June 2012. To appear.

[11] P. Stone and M. Veloso. Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork. *Artificial Intelligence*, 110(2):241–273, June 1999.

[12] D. Urieli, P. MacAlpine, S. Kalyanakrishnan, Y. Bentor, and P. Stone. On optimizing interdependent skills: A case study in simulated 3d humanoid robot soccer. In K. Tumer, P. Yolum, L. Sonenberg, and P. Stone, editors, *Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, volume 2, pages 769–776. IFAAMAS, May 2011.

Appendix

A Role Assignment Function f_v

The following is a more in depth analysis of the the role assignment function f_v described in Section 11.3.2.

A.1 Minimizing Longest Distance

Having all agents quickly reach the target destinations of a formation is important for proper strategy execution, particularly that of set plays for game stoppages discussed in Section 11.2 where there is a set time limit of 15 seconds before play resumes. It is trivial to determine that f_v selects a mapping of agents to role positions that minimizes the time for all agents to have reached their target destinations. The total time it takes for all agents to move to their desired positions is determined by the time it takes for the last agent to reach its target position. As the first comparison between mapping costs is the maximum distance that any single agent in a mapping must travel, and it is assumed that all agents move toward their targets at the same constant rate, the property of minimizing the longest distance holds for f_v.

A.2 Avoiding Collisions

Given the assumptions that no two agents and no two role positions occupy the same position on the field, and that all agents move toward role positions along a straight line at the same constant speed, if two agents collide it means that they both started moving from positions that are the same distance away from the collision point. Furthermore if either agent were to move to the collision point, and then move to the target of the other agent, its total path distance to reach that target would be the same as the path distance of the other agent to that same target. Considering that we are working in a Euclidean space, by the triangle inequality we know that the straight path from the first agent to the second agent's target will be less than the path distance of the first agent moving to the collision point and then moving on to the second agent's target (which is equal to the distance of the second agent moving on a straight line to its target). Thus if the two colliding agents were to switch targets the maximum distance either is traveling will be reduced, thereby by reducing the cost of the mapping, and the collision will be avoided. Figure 16 illustrates an example of this scenario.

The following is a proof sketch related to Figure 16 that no collisions will occur.

Assumption. *Agents A1 and A2 move at constant velocity v on straight line paths to static positions P2 and P1 respectively. A1 $\neq$ A2 and P1 $\neq$ P2. Agents collide at point C at time t.*

Claim. *A1→P2 and A2→P1 is an optimal mapping returned by f_v.*

Case 1. *P1 and P2 $\neq$ C.*
By assumption:
$\overline{A_1C} = \overline{A_2C} = vt$
$\overline{A_1P_2} = \overline{A_1C} + \overline{CP_2} = \overline{A_2C} + \overline{CP_2}$
$\overline{A_2P_1} = \overline{A_2C} + \overline{CP_1} = \overline{A_1C} + \overline{CP_1}$

By triangle inequality:
$\overline{A_1P_1} < \overline{A_1C} + \overline{CP_1} = \overline{A_2P_1}$
$\overline{A_2P_2} < \overline{A_2C} + \overline{CP_2} = \overline{A_1P_2}$

$max(\overline{A_1P_1}, \overline{A_2P_2}) < max(\overline{A_1P_2}, \overline{A_2P_1})$
$\therefore cost(A1 \rightarrow P1, A2 \rightarrow P2) < cost(A1 \rightarrow P2, A2 \rightarrow P1)$ *and claim is False.*

Case 2. *P1 = C, P2 $\neq$ C.*
By assumption:
$\overline{CP_2} > \overline{CP_1} = 0$
$\overline{A_2C} \leq \overline{A_1C} = vt$
$\overline{A_1P_1} = \overline{A_1C} < \overline{A_1C} + \overline{CP_2} = \overline{A_1P_2}$

By triangle inequality:

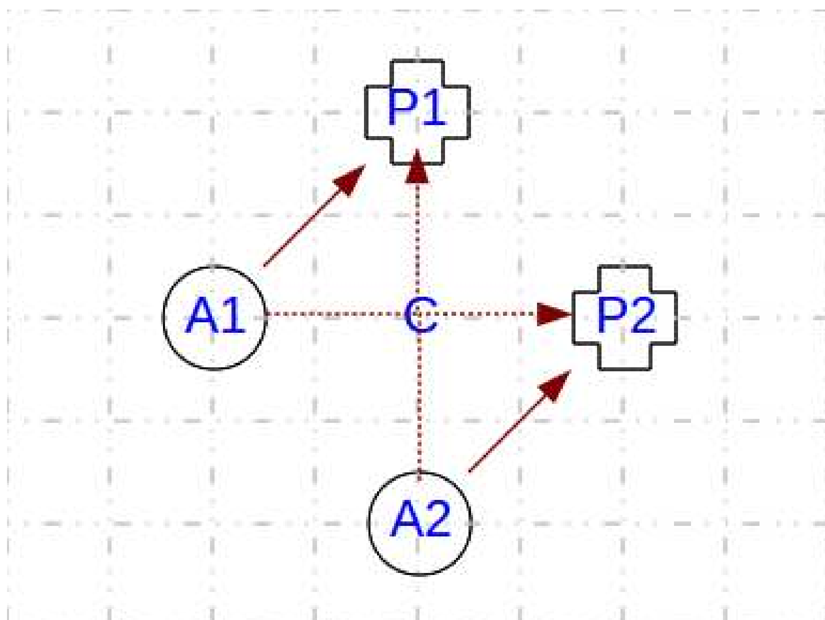

Figure 16: Example collision scenario. If the mapping (A1→P2,A2→P1) is chosen the agents will follow the dotted paths and collide at the point marked with a C. Instead f_v will choose the mapping (A1→P1,A2→P2), as this minimizes maximum path distances, and the agents will follow the path denoted by the solid arrows thereby avoiding the collision.

$if\ \overline{A_1C} = \overline{A_2C}$
$\quad \overline{A_2P_2} < \overline{A_2C} + \overline{CP_2} = \overline{A_1C} + \overline{CP_2} = \overline{A_1P_2}$
$otherwise\ \overline{A_2C} < \overline{A_1C}$
$\quad \overline{A_2P_2} \leq \overline{A_2C} + \overline{CP_2} < \overline{A_1C} + \overline{CP_2} = \overline{A_1P_2}$

$max(\overline{A_1P_1}, \overline{A_2P_2}) < max(\overline{A_1P_2}, \overline{A_2P_1})$
$\therefore cost(A1 \to P1, A2 \to P2) < cost(A1 \to P2, A2 \to P1)$ *and claim is False*

Case 3. *P2 = C, P1 ≠ C.*
Claim False by corollary to Case 2.

Case 4. *P1, P2 = C.*
Claim False by assumption.

As claim is False for all cases f_v does not return mappings with collisions. □

A.3 Dynamic Consistency

Dynamic consistency is important such that as agents move toward fixed target role positions they do not continually switch or thrash between roles and never reach their target positions. Given the assumption that all agents move toward target positions at the same constant rate, all distances to targets in a mapping of agents to role positions will decrease at the same constant rate as the agents move until becoming 0 when an agent reaches its destination. Considering that agents move toward their target positions on straight line paths, it is not possible for the distance between any agent and any role position to decrease faster than the distance between an agent and the role position it is assigned to move toward. This means that the cost of any mapping can not improve over time any faster than the lowest cost mapping being followed, and thus dynamic consistency is preserved. Note that it is possible for two mappings of agents to role positions to have the same cost as the case of two agents being equidistant to two role positions. In this case one of the mappings may be arbitrarily selected and followed by the agents. As soon as the agents start moving the selected mapping will acquire and maintain a lower cost than the unselected mapping. The only way that the mappings could continue to have the same cost would be if the two role positions occupy the same place on the field, however, as stated in the given assumptions for f_v, this is not allowed.

A.4 Other Role Assignment Functions

Other potential ordering heuristics for mappings of agents to target positions include both minimizing the sum of all distances traveled and also minimizing the sum of all path distances squared. Neither of these heuristics preserve all the desired properties which are true for f_v. As can be seen in the example given in Figure 17, none of the three properties hold when minimizing the sum of all path distances. The third property of all agents having reached their target destinations is not true when minimizing the sum of path distances squared as shown in the example in Figure 18.

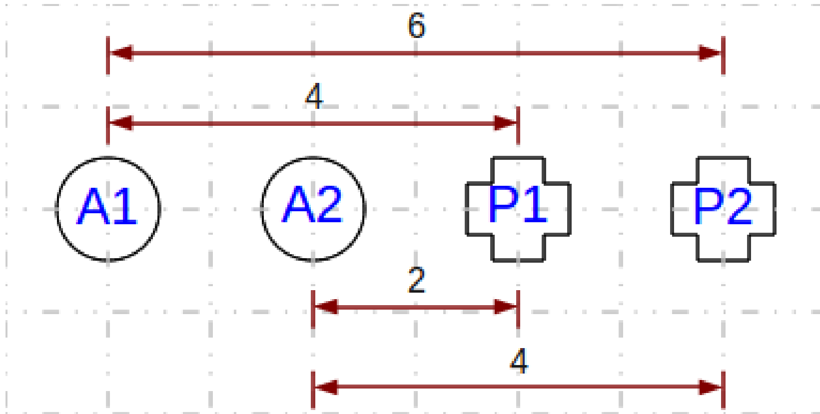

Figure 17: Example where minimizing the sum of path distances fails to hold desired properties. Both mappings of (A1→P1,A2→P2) and (A1→P2,A2→P1) have a sum of distances value of 8. The mapping (A1→P2,A2→P1) will result in a collision and has a longer maximum distance of 6 than the mapping (A1→P1,A2→P2) whose maximum distance is 4. Once a mapping is chosen and the agents start moving the sum of distances of the two mappings will remain equal which could result in thrashing between the two.

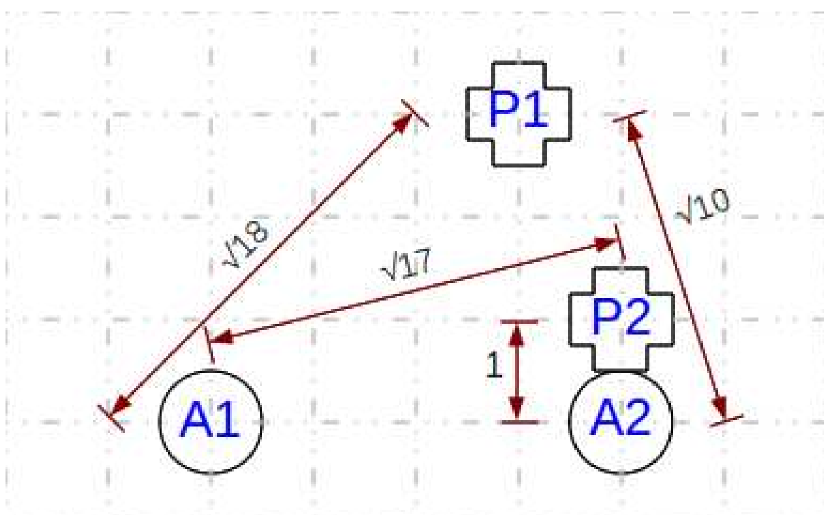

Figure 18: Example where minimizing the sum of path distances squared fails to hold desired property of minimizing the time for all agents to have reached their target destinations. The mapping (A1→P1,A2→P2) has a path distance squared sum of 19 which is less than the mapping (A1→P2,A2→P1) for which this sum is 27. f_v will choose the mapping with the greater sum as its maximum path distance (proportional to the time for all agents to have reached their targets) is $\sqrt{17}$ which is less than the other mapping's maximum path distance of $\sqrt{18}$.