The University of Pennsylvania
Robocup 2004 Legged Soccer Team

David Cohen, Yao Hua Ooi, Paul Vernaza, and Daniel D. Lee

General Robotics Automation, Sensing and Perception (GRASP) Laboratory
University of Pennsylvania, Philadelphia, PA 19104
WWW home page: http://www.cis.upenn.edu/robocup

Abstract. This paper presents the software design for a team of soccer playing robots developed at the Univ. of Pennsylvania for the 2004 Robocup competition. The software was a port and slight modification from the 2003 competition code. Lower level sensory and motor functions were first prototyped in Matlab. High level behaviors and team coordination were implemented using an embedded Perl interpreter. The robots’ walk parameters were optimized using a hill climbing routine. These developments allowed the development of the team that ultimately resulted in a fourth place finish at the international competition.

1 Introduction

Robocup 2004 marked the second year since a complete overhaul of the University of Pennsylvania team and software. The team and code were started anew early in 2003. The change proved successful, leading to a second place finish in 2003. Going into 2004 with a relatively modern and successful code base while facing the challenge of adapting to a new hardware platform in the ERS-7, the team decided to keep the greater part of its software the same, choosing to concentrate its effort on porting to the new platform and improving on weaknesses discovered in the 2003 competition.

One lesson learned from past competitions was that speed and efficiency of gait is a major, if not dominant, factor in overall competitive ability. Inspired by rUNSWift’s documented success in automatic gait optimization, our team sought to apply machine learning techniques to optimize our gait engine. In the process, no less than three completely different gait engines were implemented, and various differing approaches to gait optimization were attempted. The most successful was based on a hill-climbing algorithm and is described elsewhere in this paper.

The software architecture for the behavior-controlling state machine was also overhauled, following a more object-oriented design pattern. New kicks were designed to match the strengths of the new ERS-7. Finally, many other miscellaneous changes were made to allow our code to compile, run, and behave correctly on the new platform. Fortunately, we were able to do this in an efficient matter thanks to an overall architecture tuned for rapid development and fast parameter tuning that was inherited from last year.
2 Software Architecture

The software architecture for the robots is shown in Figure 1. This architecture is implemented by compiling several OPEN-R modules using the Sony OPEN-R API [1]:

**ROBOTCOM.BIN** Main module that includes the embedded Perl interpreter, vision and sensor interfaces, walk routine, and world model.

**EFFECTOR.BIN** Module that accepts arrays of joint angles and sequences the motors and other effectors such as LED’s. This module is largely deprecated in 2004 by new functionality in RobotComm that accomplishes the same task. However, the module is still maintained for compatibility with old interfaces.

**SNDCOMM.BIN** Module that can play and record PCM sound samples.

In order to simplify development, all interprocess communications are performed by passing a shared memory matrix structure between the modules:

```cpp
template <class T>
class ShmArray : public OShmPtrBase {
    ...
}
```

This structure allows for efficient communication as well as unifying the input and output interfaces of every module that is developed. Modules can also easily
pass information to modules on other robots by sending these data structures through the supplied TCPGateway interface.

2.1 Matlab Interface

Rapid development of lower level sensory and motor functions is facilitated by using Matlab to first prototype the algorithms [2]. Matlab is a high-level numerical scripting language that allows complex algorithms to be coded in a small number of lines. Since Matlab uses matrices as its basic data structure, we developed several functions that allowed Matlab to send and receive matrices over the wireless network to the various OPEN-R modules running on the robots.

For example, the joint angles for a new kick are first computed using a Matlab routine. This results in a $16 \times N$ matrix in Matlab that describes the motion of each of the 3 joints in the 4 legs as well as the 4 head angles (pan, tilt, roll, mouth). This matrix is then sent to the Effector module on a running robot to be actuated in real-time. These angles are then modified in Matlab and replayed on the robot to quickly improve the performance of the motion. This interface allows for the development of a new kick in only a few hours.

2.2 Perl Interpreter

Once again, the Perl programming language was used to implement high-level behaviors and for rapid development. Perl is a high-level programming language that incorporates an optimized interpreter that compiles scripts into intermediate opcodes for efficient performance [3]. We were pleased to see that more teams were employing high-level languages in their designs in 2004 than in the past. We continue to strongly advocate their use as a way to enable rapid development and eschew many of the problems encountered with programming in C++.

To enable this capability on the robots, we embedded a Perl interpreter to run on the robots [4]. Unfortunately, it is not possible to simply configure Perl for the Aibo robots using the normal build process. Due to the limited operating system environment on the robots, our Perl port is based upon the scarcely-documented microperl build in the 5.8.0 release of Perl. The interpreter is built as a static library that is linked into the ROBOTCOMM module. The resulting library references several functions such as “fork” in the standard UNIX API that are not available on the robots, so several dummy functions also needed to be implemented in order for the module to be linked properly.

To enable the Perl interpreter to interact with the sensory and motor routines implemented in the OPEN-R modules, the interpreter was extended so that it is able to call C functions exported by the modules. A small language called XS available in the Perl distribution was used to automate this task \(^1\). Through this interface, a Perl script is able to access data variables in the OPEN-R routines and activate functions that set status LED’s, sequence motions and walk routines, and communicate with other robots to coordinate team behaviors.

\(^1\) see the perlxs UNIX man page for more information
The use of the Perl interpreter allows us to develop and modify behaviors for the robots using a standard high-level scripting language in the robots’ runtime environment. Thus, we could test a new behavior by sending a new script to robots over the network and execute them on demand without having to reboot the robots. Additionally, due to Perl error handling, a bad script will rarely result in a system crash. This dramatically reduces the severity and duration of downtime during development time.

3 Vision

Most of the algorithms used for processing visual information from the robots’ CMOS cameras are similar to those used by other teams in the past [5]. Since fast vision is so crucial for the robots’ behaviors, these algorithms were implemented from scratch to gain as much performance speed as possible. This speed was critical during the competition since we used several robots with older, slower processors during the preliminary rounds.

3.1 Object Recognition

The main processing pipeline involves segmenting the highest-resolution color images from the camera, forming connected regions, and recognizing various objects from the statistics of the colored regions. The color segmentation routine classifies individual pixels in the image based upon their YCbCr values. Based upon a number of training images, a Gaussian mixture model is used to segment the YCbCr color cube into the following colors:

- Orange (Ball)
- Pink (Marker)
- Cyan (Marker and Goal)
- Yellow (Marker and Goal)
- Blue (Robot)
- Red (Robot)
- Green (Field)
- White (Lines and Border)

Once the pixels in the image are classified according to their colors, they are merged into connected components using techniques that have been previously described [5]. This is accomplished by first run-length encoding the images, and then merging these run-lengths into connected regions.

After the image has been segmented into these connected regions, the regions are classified into relevant objects by comparing various image statistics of the regions. These statistics include the bounding box of the region, the centroid location, and the major and minor axes lengths. In this manner, the location of the ball, markers, and goals are detected.

An unfortunate deficiency in our vision system is the ability to detect other robots. Since accurate object recognition by image still lacks a computationally
efficient solution in the general case, this is a difficult problem that admits no simple solution. We hope to examine insights and workarounds from other teams in order to improve on this in 2005.

### 3.2 Line Recognition

Our code featured the same line recognition algorithm as it did in 2003. Field line recognition decreases the need for our robots to actively search for field markers, enabling them to chase the ball more effectively. The first step in line identification is to find white pixels in the medium resolution camera images that neighbor pixels of field green color. Once these pixels are located, a Hough transform is used to search for relevant line directions.

In the Hough transform, each possible line pixel \((x, y)\) in the image is transformed into a discrete set of points \((\theta_i, r_i)\) which satisfy:

\[
x \cos \theta_i + y \sin \theta = r_i
\]

![Fig. 2. Hough transformation for field line detection in images.](image)

The pairs \((\theta_i, r_i)\) are accumulated in a matrix structure where lines appear as large values as shown in Figure 2. To speed the search for relevant lines, our implementation only considers possible line directions that are either parallel or perpendicular in the field to the maximal value of the accumulator array. Once these lines are located, they are identified as either interior or exterior field lines based upon their position and then used to aid in localization.

### 4 Localization

The problem of knowing the location of the robots on the field is handled by a probabilistic model incorporating information from visual landmarks such as markers, goals, and lines, as well as odometry information from the effector module [6]. Recently, probabilistic models for pose estimation such as extended Kalman filters, grid-based Markov models, and Monte Carlo particle filters have been successfully deployed. Unfortunately, complex probabilistic models can be
difficult to implement in real-time due to a lack of processing power on board the robots. We address this issue with a new pose estimation algorithm that incorporates a hybrid Rao-Blackwellized representation that reduces computational time, while still providing for a high level of accuracy. This new algorithm models the pose uncertainty as a distribution over a \textit{discrete} set of heading angles and \textit{continuous} translational coordinates. The distribution over poses \((x, y, \theta)\), where \((x, y)\) are the two-dimensional translational coordinates of the robot on the field, and \(\theta\) is the heading angle, is first generically decomposed into the product:

\[
P(x, y, \theta) = P(\theta)P(x, y|\theta) = \sum_i P(\theta_i)P(x, y|\theta_i)
\]  

We model the distribution \(P(\theta)\) as a discrete set of weighted samples \(\{\theta_i\}\), and the conditional likelihood \(P(x, y|\theta)\) as simple two-dimensional Gaussians. This approach has the advantage of combining discrete Markov updates for the heading angle with Kalman filter updates for the translational degrees of freedom.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{hybrid_representation.png}
\caption{Hybrid probabilistic representation used for localization.}
\end{figure}

When the algorithm is implemented on the robots, they are able to quickly incorporate visual landmarks and motion information to consistently estimate both the heading angle and translational coordinates on the field as shown in Figure 3. Even after the robots are lifted (kidnapped) by the referees, they are able to quickly relocalize their positions when they see new visual cues.

5 Motion

The motion of the robots is controlled by a parameterized walk routine in addition to predetermined scripted motions. For the RoboCup 2004 competition, the UPennalizers implemented a machine learning system for gait optimization, similar to that of the Univ. of Texas [9]. However, the system differed from theirs in two key respects. First, the walk optimized was not stiffly geometric but was,
instead, a dynamic parameterized C walk code. Second, a watcher-walker system was used instead of a system whereby the walker self-localized using beacons; we explain later why this method was preferred.

Future work might include incorporating an overhead camera into the system, thereby eliminating most of the measurement noise of the system. More sophisticated algorithms can then be applied. Also, a system for tuning more agile walks should be considered before the 2005 competition.

5.1 Dynamic parameterized C walk code

The incorporation of the C Walk Code was, in fact, incredibly simple due to the Perl machine interface. The prime difficulty was in making what was inherently an unsteady (but speedy) walk consistently stay on track between two markers: this involved a large amount of tweaking to the agent code to run well consistently (even with the tweaks, running experiments required constant supervision).

The parameters tuned in this case were:

1. Tilt (rad)
2. Height (mm)
3. Fore X offset (mm)
4. Fore Y offset (mm)
5. Hind X offset (mm)
6. Hind Y offset (mm)
7. Max Distance per frame (mm)
8. Fore Lift initial (mm)
9. Fore Lift final (mm)
10. Hind Lift initial (mm)
11. Hind Lift final (mm)
12. Fore X minimum (mm)
13. Fore X maximum (mm)
14. Fore Y minimum (mm)
15. Fore Y maximum (mm)
16. Hind X minimum (mm)
17. Hind X maximum (mm)
18. Hind Y minimum (mm)
19. Hind Y maximum (mm)

Since the walk engine features an active/recovery area system, the parameterization is slightly different: basically, ellipses are identified (using parameters 3-6 and 12-19) wherein the robot performs its steps. Once the foot leaves these ellipses, it travels through a recovery space determined by the position of the feet already on the ground (so as to maintain balance) and parameters 8-11. The general position of the robot’s torso is determined with parameters 1 and 2 and parameter 7 determines how far the robot’s legs are allowed to move per frame.
5.2 Watcher-walker system

An avenue for future work noted during early, self-localizing runs was the minimization of measurement noise through modifications to the agent code. It was decided that the walk learning would follow this track, specifically with respect to making a portable, low-noise system for modifying walks on site with results that would be immediately portable to the game code. The decision was made to implement a watcher-walker system, wherein an observer dog would watch the walker dog test gaits (generated in C instead of PERL) and return refined displacement estimates. It will be shown that this system, while imperfect, leads to less error than the system employed prior (which had the walker self-identifying its localization) and has the added benefit of properly positioning the head during testing (leading to better balance during gameplay).

In order to justify this claim, we need to derive estimates for how localization is affected by slight vision errors. In order to derive those estimates, we need to construct models of how the robots are modeled in both the self-localizing and the watcher-walker systems.

Fig. 4. Schematics of the Self-Localizing system and the Watcher-Walker setup.

In order to estimate the sensitivity of our distance estimates to noise in our angle estimates, we find the percentage change in our distance estimate given
a change in measurement angle, calculated as \((1/L)(dL/d(\text{angle}))\). For the self localizing case:

\[
L_1 = D_1 / \tan(\theta)
\]

\[
(x/L_1)dL_1/d\theta = -(x/L_1)(D_1 / \tan^2(\theta))(1 + \tan^2(\theta))
\]

For the watcher-walker case:

\[
L_2 = D_2 \tan(\phi)
\]

\[
(x/L_2)dL_2/d\phi = (x/L_2)(D_2 / \cos^2(\phi))
\]

Given approximate real-world values: \(L_1 = 3\) m, \(D_1 = 0.5\) m, \(\theta = \tan(D_1/L_1) = 0.1651\), \(L_2 = 1.5\) m, \(D_2 = 2\) m, \(\phi = \tan(L_2/D_2) = 0.6435\), we can derive the sensitivities for the two cases:

\[
(x/L_1)dL_1/d\theta = -(x/L_1)(D_1 / \tan^2(\theta))(1 + \tan^2(\theta)) = -6.1703
\]

\[
(x/L_2)dL_2/d\phi = (x/L_2)(D_2 / \cos^2(\phi)) = 2.0833.
\]

Therefore the self localizing dog has a sensitivity to angular perturbations about three times larger in magnitude than his friend who has a watcher\(^2\). There is also reason to expect that a dog who is bouncing around while turning will have larger noise in its measurement angle than his friend, who is completely dedicated to watching. Therefore, there is some theoretical basis to developing a watcher-walker system.

The implementation of the watcher was very simple: a ball was affixed to the top of the walker, and the watcher was set down and told, in short, to “watch the ball”. His ball estimates were distributed (automatically) over wireless to the walker, who updated his localization estimate appropriately.

### 5.3 Walk results

Over 16 iterations, the dog’s walking speed increased from \(\sim 330\) mm/sec to about 415 mm/sec (with a maximum estimated speed of 430 mm/sec). This is comparable to the top speeds exhibited in RoboCup 2004. This final policy exhibited the maximum speed of 430 mm/sec, and was placed directly on the game stick, with good results.

Some information can be gleaned from the way the parameters behaved over the 16 runs. In particular, a statistical analysis can be performed to try to determine what parameters have the most effect on the walk speed. For each parameter, we consider a null hypothesis which is “the learning method is indifferent as to the value of this parameter.” In order to reject this null hypothesis, we have to first construct a probability distribution for the end value of a variable that, starting from zero, for \(n\) (16 in our case) iterations increases by 1, decreases by 1, and remains constant with equal probability. This is a deceptively difficult problem, but it can be simulated very easily in MATLAB. Our statistical analysis showed that with 90% certainty, we can reject this null hypothesis for 7 parameters:

- Tilt (absolute change 8)

\(^2\) For the case where \(L_2\) is 1.5 m, this sensitivity reaches a minimum when \(D_2\) is 1.5 m.
– Fore Y offset (absolute change 6)
– Hind X offset (absolute change 5)
– Fore Lift initial (absolute change 5)
– Fore Lift final (absolute change 6)
– Fore X maximum (absolute change 5)
– Hind X minimum (absolute change 5)

It should not be inferred that the other parameters are irrelevant, but this is an indication of which current parameters are most important for tuning. Avenues for future work include the development of a fast “agile” gait (to be tuned on an obstacle course of sorts), the application of more sophisticated learning algorithms, and the incorporation of overhead camera data.

5.4 Gait

In order to complement our team’s fast and highly reactive style of play, we designed an innovative gait algorithm that features the ability to change trajectory at virtually any time. This is in contrast to standard approaches to gait generation that rely on periodic motions that only allow trajectory changes after each step or half-step.

The design philosophy that allows this change is simple: instead of having foot position be a function mainly of time, we have foot position as a function principally of current position and desired trajectory. At every time step (new camera frame), a new trajectory is calculated, and new feet positions are calculated by generating appropriate translations of the current feet positions. A check is then performed to determine whether the feet fall inside a predefined workspace. If they are outside this region, the legs are lifted, and a recovery stroke is performed. Otherwise, they are immediately moved to the new desired positions.

The gait is parameterized by a large set of parameters that can be changed at any time—including in the middle of a step. This allows us to adapt to different play situations very quickly. The tunable parameters are also very useful for interfacing with automatic gait optimization routines.

5.5 Kicks

The kicks are fully scripted motions, each starting from a standard position that is compatible with the end of a walk cycle. Matlab was used to tweak the joint angles in order to develop a set of kicks that could be used to kick the ball forward as well as sideways using both the legs and head. The criteria for developing kicks were as follows:

1. They shouldn’t damage the robot or turn it off under any circumstances.
2. They had to be very quick to execute (typically under 2 seconds).
3. They featured complete ball control at every instant until the ball was released.
4. If they failed to release the ball, they should return to a state where the robot maintained control of the ball.
5. They were optimized for either power or accuracy.

The final set of kicks used in the competition were quite successful in scoring from almost every position on the field, as well as knocking the ball from the other team’s possession.

6 Behaviors

The behaviors of the robots on the field are implemented using an object-oriented Perl script. This script incorporated an event-driven state machine that generates different actions depending upon one of possible four roles that the robot was assigned:

**Attack** The attacking robot goes directly to the ball, as long as it wasn’t in the defensive penalty box.

**Defend** The defending robot positions itself between the ball and defensive goal area.

**Support** The supporting robot positions itself in an area away from the ball that would assist in scoring.

**Goalie** The goalie stays near the defensive goal to clear the ball when it comes close.

Though the overall idea remained the same, the implementation of the behavior state machine was overhauled in 2004. The prior implementation featured a monolithic structure that kept all of the behavior code for a certain role in a single file. This made it unwieldy and difficult to factor code for common use between player and goalie state machines.

The solution is a new object-oriented architecture. Each state is represented by a unique class implementing a common interface for states. When a state transition occurs, the new state class is instantiated, and a reference to the object is stored in a controlling state machine object. The state machine object can then invoke the interface methods on the currently-stored state to perform tasks specified by the state. This approach provides a structure that is very clean, modular, and easy to maintain.

6.1 Potential Fields

Potential fields are used in all the roles to guide the robots to their optimal positions on the field. As shown in Figure 4, the positioning of the robots are dependent upon the ball location relative to predetermined field locations. For example, the optimal position for the Support role is on an intermediate point between the ball and an offensive field position. Similarly, the Defend role is situated on an intermediate point between the ball and a defensive field position. With the exception of the Goalie, a repulsive potential field in the defensive penalty area prevents the robots from becoming an “illegal defender.”
6.2 Role Switching

Although the Goalie remains statically assigned, the three robots that are initially assigned the Attack, Support, and Defend roles can fluidly switch roles. Any of these three robots can actively switch to the Attack role if it clearly sees the ball and if it is not being suppressed from switching by a timer. Once a new robot switches to an Attack role, it sends a message over the network to the other two robots who take either a Support or Defend role based upon their relative position. The more forward positioned robot becomes Support, and the backward positioned robot takes the Defend role. At this point, the suppression timer on the Support and Defend robots are reset so that they cannot immediately take the Attack role.

As long as the Attack robot sees the ball and is making progress towards it, it will periodically send messages to the other robots to reset their suppression timers. By adjusting the values of the timer periods based upon relative distance to the ball, it is possible to ensure that the nearest robot to the ball ultimately becomes and stays the Attack robot.

Once again, unusual network latency due to the large amount of radio interference at Robocup exposed unfortunate deficiencies in this role-switching scheme. Distributed algorithms more appropriate to this scenario are sought to address this problem. Additionally, it is anticipated that improved vision routines will help address this by offering direct observation of fellow robots.
7 Summary

Whereas Penn’s 2003 team was characterized by an emphasis on infrastructure-building and rapid development, 2004 was marked by a concentration on gait algorithms and porting issues. The work proved fruitful, as our gait was subjectively at least as effective as those of the top teams in 2004. In retrospect, however, more work could have been done on sensory and coordination algorithms. The lack of a few key features such as the ability to identify other robots by vision, were important factors that prevented us from being able to truly compete on an equal ground with the top finishers. It is hoped that we will be able to improve on these aspects in 2005 by examining and building on the work of those pioneering teams. Likewise, it is hoped that our code, released openly under the GNU Public License, will also be of use to future teams.

References