CMPack'04: Team Report

Manuela Veloso, Paul Rybski, Sonia Chernova, Douglas Vail, Scott Lenser, Colin McMillen, James Bruce, Francesco Tamburrino, Juan Fasola, Matt Carson, Alex Trevor

Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213
http://www.cs.cmu.edu/~coral

1 Introduction

The CMPACK'04 team follows our past teams CMPack'03, CMPack'02, CMPack'01, CMPack'00, CMTrio'99, and CMTrio'98 [7, 5, 10, 9]. In our research this year, we explored the capabilities of the new ERS-7 robotic platform; developed new ways of modeling the world, such as using accelerometer data to detect when the robot is entangled; and maintained our focus on robust behaviors and cooperation. In this paper, we will focus in detail on several recent components of the team.

2 Constructing a Local Model

The vision module is responsible for converting raw YUV camera images into information about objects that the robot sees and are relevant to our task. This is done in two major stages. The low level vision processes the whole frame and identifies regions of the same symbolic color. We use CMVision [2, 1] for our low level vision processing. The high level vision uses these regions to find objects of interest in the image.

2.1 Radial Vision

The vision component is responsible for detecting objects present along different angles around the robot. The vision processing consists of several discrete stages. The first stage takes the raw camera image and classifies each pixel into one of several color classes. Pixels corresponding to the floor are classified into the "floor" class. Pixels of other colors are classified into either one of the color classes for various objects or into the "unknown" class for general obstacles. Scan lines are then drawn in the image that correspond to lines on the ground plane emanating from the reference point for the robot. Figure 1 shows an example image with the scan lines drawn over the image. We used scan lines spaced every $5°$ degrees around the robot. The classified pixels on each scan line are then run length encoded. Objects are located along each scan line and identified if possible. Most of the run time is spent classifying the image which does one table lookup per pixel. Other parts of the processing contribute negligible additional time. The algorithm does an excellent job of locating objects and identifying them when possible. Figure 2 gives pseudo-code for the complete algorithm.

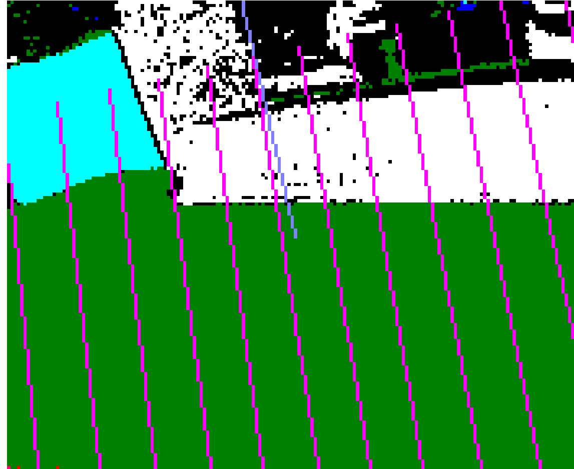


Fig. 1. Radial vision image with scan lines.

Segmentation: We segment the image into color classes using CMVision 2, a real time color vision library [2]. CMVision uses a lookup table(threshold map) to perform the mapping from YUV pixel values to symbolic color class. We use 4 bits of Y and bits of U and V to make a 16 bit index into the lookup table. Each entry in the table has the symbolic color class that pixels of that image pixel color should be assigned. Note that each symbolic color class may correspond to several physical colors, e.g. a "floor" color class which includes all the colors found on the floor. The threshold map is trained by taking images with the camera and hand labelling the symbolic color classes to be used for that image. The color classes we use include a color class which corresponds to unknown colors. We take advantage of this class to identify obstacles of unknown colors which are known to not be of the same color as the floor (the "floor" color class). The camera and hand labelled images define a supervised classification problem. We take an example driven approach to this problem.

Our solution is to take each example pixel in YUV space and spread it using geometric decay in all directions of the full YUV space. The geometric decay is done in Manhattan distance which allows for an efficient implementation using a fill stage followed by a constant number of sweeps across the threshold map. We use a geometric decay rate of .5 which decays rapidly with distance. For each box in the threshold map, the total example weight due to each color class is calculated. Each color class has an associated confidence threshold. If the proportion of weight due to a particular color class compared to the total weight in this threshold box is greater than the confidence threshold of the color class, the threshold box is labelled with this symbolic color. If no color class satisfies these conditions, the threshold box is labelled with the "unknown" color class.

Object detection: The vision identifies the following object types: *wall*, *stripe*, *unknown_obstacle*, *red_robot*, *blue_robot*, *ball*, *cyan_goal*, and *yellow_goal*. All of the ob-

```
Procedure Vision(v:vision)
  Project corners of image into egocentric coordinates.
  Calculate angle of each projected corner.
  let min_angle ← Minimum angle of corners.
  let max_angle ← Maximum angle of corners.
  Snap min_angle and max_angle to one of the
    fixed scan angles.
  for α ← min_angle to max_angle
    Find the line on the image l which corresponds to
      a line on the ground at angle α.
    Scan through the segmented image along l creating
      a run length encoded version of the line r.
      Store the start/end screen coordinates of each run.
    IdentifyObjects(v,AngleIdx(α),r)
```

Fig. 2. Radial vision

jects are identified using the following heuristic: the object is the first continuous set of pixels in the scan line composed completely of color classes that are "ok" for the object and which has k more pixels of the "best" color class for this object than all other color classes. The value of k reflects the amount of noise present in the image; we use a value of 5. Since *wall* and *stripe* are both composed of the same color (white), an additional test is needed to disambiguate when a *wall/stripe* is detected. The object is considered a *wall* only if it is at least 50mm wide and the number of white pixels on the *wall* is at least as great as the number of green pixels after the *wall*. The first filter makes sure the wall could not just be a *stripe*. The second filter is due to our use of a green carpet as our floor which does not extend much beyond the white walls. These filters reliably distinguish between *wall* which must be avoided and *stripe* which can be transversed safely. An *unknown_obstacle* is detected whenever enough pixels belonging to the "unknown" class occur together. Since one of the color classes is "floor", this corresponds to occlusions of the floor by other objects. These objects get labels as *unknown_obstacles* in this manner. Because the bodies of robots are also of unknown color, *unknown_obstacles* often occur for us due to robots. We peek ahead along this scan line to see if this obstacle can actually be identified and if so move this obstacle into the *red_robot* or *blue_robot* class as appropriate.

Distance to objects is calculated by intersecting rays through the closest pixels of the object on the image onto the ground plane the robot is standing on. This generates very accurate distances for objects that are close and noisy estimates for more distant objects. Because only the closer objects are useful for obstacle avoidance, distances to the objects of interest are all quite accurate. Some distances can be slightly high due to parts of the object being off of the ground. This also is not a problem because the estimates improve somewhat as the robot approaches. Since objects in the air must be supported by something, the supports are usually also visible and since they are resting

on the ground, they will have accurate distance estimates. So for all objects of interest for obstacle avoidance, accurate distance estimates are available.

2.2 Local Model

Since the camera has a limited field of view (55° in the case of our robot), it is necessary to keep a model of the local environment to ensure good obstacle avoidance performance. The purpose of the model is to remember objects that the robot has seen recently that must be avoided and are currently outside the field of view of the camera.

The model we present here provides an alternative to the common occupancy grid model.

The model is based upon an odometric reference frame. This reference frame is immobile with respect to the robot. So the robot moves in the reference frame; the reference frame does not move as the robot moves. This is not the same as a global reference frame, though, as no attempt is made to make particular coordinates in the reference frame correspond to points in the robot's environment. Indeed, it is expected that during normal operation, the reference will slowly drift with respect to world coordinates due to the accumulation of odometric error. This is all that is needed for modelling local obstacles. Syncing the reference frame with the global world reference frame is superfluous. With this reference frame, updating the model for the movement of the robot simply requires the position of the robot within the reference frame to reflect the motion of the robot. This is an extremely fast operation.

The model uses a point based representation of the objects in the robot's vicinity. Each point represents a single observation of an object in the environment. Each point stores the following information:

- The location of the point in the odometric reference frame.
- The type of object observed.
- The time at which the object was observed.

These points are stored in a spatial data structure for quick access. Updating the model for vision simply inserts the current observations (projected into the odometric reference frame) into the model/spatial data structure. This is a fast update to perform, so the overall model can be up to date with very little computational overhead.

Old data in the model is aged in order to make sure that the memory and computation resource usage is bounded. Data after a certain amount of time is pretty useless anyway due to the accumulation of odometric error and the movement of objects in the environment. To support this bounded nature, data older than a certain age is simply discarded. This operation can be implemented very efficiently as described below.

The storage of the points in the model is based on standard spatial data structure techniques. The points are stored in a binary tree with axis aligned splitting planes that divide the remaining space in half with each step. It is easy to add points to this data structure efficiently. Points can also be added that are currently outside the range of values covered by the tree by growing the tree up from the old root. In order to efficiently support the discarding of old data, a generational model is adopted. The spatial tree is split into k independent spatial trees based upon the age of the data. Data is always

added to the newest tree. When the newest tree has been around long enough, a new tree is added and the oldest tree is discarded. We used $k = 4$ which provided a good balance between acceleration of data access and granularity of the discarding process for old data points. The data points were stored in an array which the spatial trees indexed. This setup resulted in a faster implementation. In our current implementation, the removal of old data points takes time $O(n/k)$ with a very low constant factor where n is the total number of points stored in the model. With some effort, this can be reduced to $O(1)$ time by intelligent indexing of the start and end indices of data points and tree nodes and using arrays for the backing storage.

In order for the model to be useful, it is necessary for the model to be able to answer useful queries. All of the queries in our model are based on orientated rectangles. A query area is determined by 5 parameters of the rectangle: the x,y location of the center, the width, the height, and the orientation. These are all specified relative to the robot and converted into the odometric reference frame to perform the actual query. The query returns all the points that fall within the query rectangle along with there position in egocentric coordinates for analysis. The analysis must use this information to answer useful queries. A wide range of possible questions can be answered based upon this facility for data lookup. In particular, we have built two types of queries. The first query (the obstacle query) reports the probability that a particular rectangle contains an obstacle and the probability that different objects are present in the query rectangle. The second query (the separator query) calculates a splitting plane for which all obstacles inside the rectangle are on the far side of the splitting plane. These two queries can be used to construct a variety of obstacle avoidance systems.

The obstacle query weights points based upon how old they are. Points older than 3 seconds are ignored. Other points get the weight $.1^t$ where t is the age of the point in seconds. Clear points from the same camera frame as obstacle points are ignored as these are usually just clear areas in front of obstacles. Each object type is assigned a probability of occurrence equal to the proportion of the total weight due to points of that object type. The probability of obstacle is simply equal to the sum of the probabilities of object types that are obstacles. The obstacle query also reports the total weight found in the rectangle. This total is used by behaviors as an indicator of the amount of data available for the query rectangle. Rectangles below a minimum total weight threshold are considered unknown. Rectangles above the total weight threshold are thresholded on the obstacle probability to decide whether they are obstacles or not.

The separator query calculates a line for which all obstacle points within the rectangle fall on the far side of the line from the robot. The query first stores all the points that fall in the query rectangle as returned by the low level query interface. The separator tries k different directions equally spaced amongst all possible angles. These k directions are interpreted as possible normal vectors for the separator line. For each direction, the separator line with that normal direction is calculated which maximizes the distance from the robot while keeping all obstacles on the far side of the line. Out of these k proposed separators, the one which has the maximum area of the query rectangle on the free side of the separator line is chosen. For convenience, this calculation is approximated by using a circular approximation of the query rectangle. The resulting

separator line can be used for finding a conservative bound on the distance from the robot to an obstacle.

3 World Model

The CMPack04 world model estimates the positions of the ball, opponents, and teammates on the soccer field and provides a single unified interface to the behaviors. Uncertainty estimates for each of these quantities are calculated as well. Each robot communicates world model state information to each of its teammates at a rate of 8Hz. The data that each robot communicates includes its own global position (computed from the localization subsystem), its best estimate of the ball position, and the 3 most recent observations of opponent robots.

3.1 Ball Tracking

The ball estimate is provided by a bank of Kalman Filters, similar to the Multiple Hypothesis Tracking Kalman Filter (MHTKF). The robot's own estimate of the ball is kept separate from the estimate returned by its teammates. When a behavior requests the position of the most likely ball, the model (self or teammate) with the least uncertainty is chosen and that position and uncertainty estimate is returned to the behavior.

Kalman Filter The Kalman filter is an example of a Bayesian filter that uses a two-step protocol to provide an estimate as to the position of the tracked object. The two steps include state *propagation* step, where the estimated position of the tracked object is updated according to its kinematic model, and the sensor *update* step, where a (noisy) sensor reading is used to update model. In both steps, both the state estimate as well as the uncertainty associated with the state are updated.

Propagation step

$$\begin{aligned} \hat{x}_{t+1/t} &= F\hat{x}_{t/t} + Bu_t \\ P_{t+1/t} &= FP_{t/t}F^T + GQ_tG^T \end{aligned}$$

Update step

$$\begin{aligned} \hat{z}_{t+1} &= H\hat{x}_{t+1/t} \\ r_{t+1} &= z_{t+1} - \hat{z}_{t+1} \\ S_{t+1} &= HP_{t+1/t}H^T + R \\ K_{t+1} &= P_{t+1/t}H^T S_{t+1}^{-1} \\ \hat{x}_{t+1/t+1} &= \hat{x}_{t+1/t} + K_{t+1}r_{t+1} \\ P_{t+1/t+1} &= P_{t+1/t} - P_{t+1/t}H_{t+1}^T S_{t+1}^{-1} H_{t+1} P_{t+1/t} \end{aligned}$$

The above notation is defined below as:

$\hat{x}_{t/t}$	estimated state vector at time t using sensor update at time t
F	state transition function
B	control function
u_t	control input at time t
$P_{t/t}$	covariance matrix at time t using sensor reading at time t
G	noise input function
Q_t	process noise covariance matrix at time t
z_t	sensor reading at time t
$\hat{z}_t$	estimated value of sensor reading at time t
H	sensor function
r_t	residual (error) between expected and actual sensor readings
R	sensor noise covariance
S_t	Computed covariance of sensor reading at time t
K_t	Kalman gain at time t

Multiple Hypothesis Tracking A shortcoming of the Kalman Filter algorithm is that it assumes that all of the noise models can be estimated using white Gaussian noise. The final state and uncertainty estimate are also represented as a single Gaussian distribution. An extension to this algorithm is the Multiple Hypothesis Tracking Kalman filter by which a multi-modal probability density can be estimated from a bank of Kalman filters.

For every object to be tracked, there exists a bank M of k Kalman filters.

1. Propagate each filter in parallel
2. Compare new sensor update against each possible hypothesis
3. Compute the Mahalanobis distance of the reading for each filter
4. If the filter with the lowest distance has a distance less than a gating threshold, choose it
5. If no filter matches the gating function, initialize a new filter with the reading
6. Prune filters with covariance that are too large

The Mahalanobis distance of a sensor reading with the given state is calculated as $r_t S_t r_t^T$. The filter M_* to accept the receive the update is selected as follows:

$$M_* \begin{cases} M_i & \text{if } M_i < 3, \\ \text{Initialize a new filter} & \text{otherwise} \end{cases}$$

Finally, a check is performed to determine whether the covariance of the estimate has grown too large to be practical. In this case, if the covariance essentially covers the entire field, then a particular filter is no longer informative (it is essentially a uniform density distribution) and is removed from consideration.

Combining Ball Readings from Teammates Each robot communicates the most likely Kalman filter model M_* from its own world model to every one of its teammates. When a robot receives multiple ball estimates, it must merge them into a single (though potentially multi-modal) density estimate using the MHTKF formalism above. Each observation consists of the mean and covariance estimate. These quantities are used as the observation and sensor covariance values for the update phase.

3.2 Ball Trajectory Estimation

The World Model contains a method that uses linear regression on the past history of ball positions to estimate the current velocity of the ball. The primary use of ball trajectory estimation is to determine appropriate times for the robot to dive or block in order to capture the ball. The procedure takes two arguments, *minHistLen* and *maxHistLen*, which control how many entries in the history of ball positions are used in the regression. Typical values are 10 and 25 respectively. For the goalie, this behavior is especially important when blocking shots on goal. In this case the *minHistLen* is set to 5, because we prefer the goalie to possibly dive spuriously (in those cases where limited data creates a false impression that the ball is moving toward the goalie) than to not dive when the ball indeed moving toward the goal and the goalie has not yet seen it for very long.

The trajectory estimation is done by performing multiple linear regressions, with varying amounts of ball history being used for each regression. The chi-squared goodness-of-fit values of the linear regressions are used to select the best one. The velocity, intercept, and χ^2 values calculated by this regression are then used in the implementation of behaviors.

This procedure presupposes the existence of a *fit()* method that takes in two arrays and a length i and uses the most recent i entries of these arrays to perform a linear regression that returns the a and b values for the line $y = ax + b$ and the chi-squared values. We used the implementation of *fit()* provided in the book *Numerical Recipes in C*. The license of this book prevents us from distributing the source code for the *fit()* function, so for our code release we replaced *fit()* with a dummy function that simply returns the current position of the ball. Typing in the code for this function verbatim from the book should be sufficient to provide the needed linear regression functionality; otherwise any compatible implementation of linear regression can be used.

3.3 Opponent Robot Tracking

Each visual observation of an opponent robot is stored for a maximum of 4 seconds after which time it is removed from consideration. Behaviors can query arbitrary rectangles in space for the existence of opponent robot observations. When a robot receives robot observations from its teammates, these observations are incorporated to the robot's own set of readings and are queried in the same fashion. Similarly, the observations are removed from consideration after four seconds.

4 Surface and State Detection from Accelerometer Data

We introduce our surface and state detection algorithm and discuss the tradeoffs in its design. We apply the algorithm to two problems: detecting when a robot is entangled with a wall or another robot and, to test the robustness of the algorithm, to identifying the surface under the robot as it walks. We present experimental results showing the robot's performance discriminating between a cement floor, the carpet in our laboratory, and the carpet on a RoboCup field. Figure 3 shows sample accelerometer data from the two carpeted surfaces; the algorithm must distinguish between these two classes of data. We present additional results where the robot detects if it is freely playing soccer, bumping into a wall, or entangled with another robot. Additional details may be find in [8].

4.1 Algorithm Details

Surface and state detection is a classification problem. We would like the robot to identify characteristics of its environment based on accelerometer data as it walks. We accomplish this through supervised learning; we require sample data from each state so that we can train our classifier.

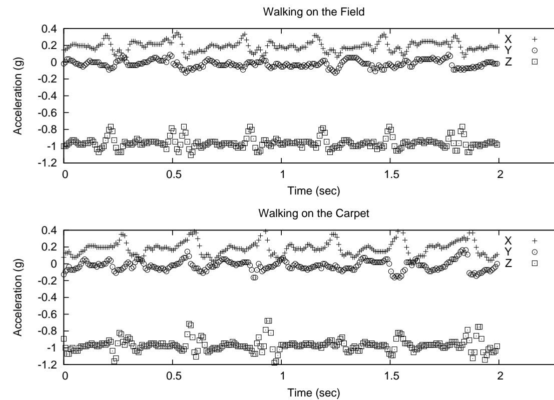


Fig. 3. Two seconds of accelerometer data gathered while the robot marched in place on two carpeted surfaces. The acceleration along the x axis has a positive mean because the robot's stance leans forward slightly. The classification algorithm must disambiguate these two signals.

Since accelerometer data arrives as a stream, we window it into discrete chunks when creating the feature vector for the classifier. Larger data windows increase accuracy at the cost of increased latency when detecting change. We chose to use a one second sliding window which, given the sensor frame rate of the robot, includes 125 accelerometer readings.

To cut down on the amount of data passed to the learner, we used statistics to described the sensor reading distribution in each window instead of the data window itself

as the feature vector. We used the variance in x, y, and z accelerations as well as the (x, y), (x, z), and (y, z) correlation coefficients over each window as the six features. The correlation coefficients were calculated as:

$$cor(x,y) = \frac{\sum_{i=1}^{n}(x_i-\overline{x})\cdot(y_i-\overline{y})}{\sqrt{\sigma(x)\cdot\sigma(y)}}$$

We chose to use decision trees for learning due to their simple representation and classification speed. Labeled training data was passed to the C4.5 decision tree learning program to generate the tree [6]. Every sensor frame when new accelerometer data becomes available, the sliding window is shifted and the decision tree is used to generate a new surface classification.

As an aside, it is possible to calculate each of the six features iteratively so that updates only require calculations involving the oldest accelerometer data point and the newest data point. Coupled with the speed of the decision tree this makes it practical to run the classifier at the sensor frame rate. Coupled with the relatively small size of the data window (125 samples, or one second worth of data), the state detection has a low enough latency to be useful to the robot as it executes behaviors.

4.2 Experimental Results

Table 1 shows the results when the classifier is trained to recognize different states that occur in a soccer game. The states considered are when the robot is freely playing, i.e. walking, kicking, and turning; when the robot is running into a wall; and when a robot is entangled with another robot. The classifier is less accurate when disambiguating between the two collision states than determining when the robot is freely playing.

To test our surface prediction algorithm, we gathered five minute segments of accelerometer readings while the robot marched in place on each of three surfaces. We created a decision tree using C4.5 and used 10-fold cross validation to quantify the accuracy of our final classifier. That is, we divided the data into ten 30 second segments, trained a decision tree using 9 of the segments and evaluated the tree on the remaining, unseen segment. We repeated this until each segment had been used to evaluate a decision tree trained on the other nine data segments.

The evaluation results are presented in table 1. As we would expect, the classifier has more trouble disambiguating between the two carpeted surfaces as they are very similar and produce similar accelerometer data. See figure 3.

Table 1. Surface Recognition and State Detection

	Classified as:		
True Class	Cement	Field	Carpet
Cement	91.0%	3.1%	5.9%
Field	4.8%	81.1%	14.1%
Carpet	7.1%	11.7%	81.2%

Overall Performance	
Correct	84.9%
Incorrect	15.1%

	Classified as:		
True Class	Playing	Hooked	Wall
Playing	97.8%	1.8%	0.4%
Hooked	1.4%	92.4%	6.2%
Wall	0.3%	15.7%	84.0%

Overall Performance	
Correct	92.3%
Incorrect	7.7%

5 Walk Learning

Speed is a critical factor in the RoboCup competition. For this reason we developed an autonomous approach for optimizing fast forward gaits based on genetic algorithms [3]. Our approach proved to be very effective on the ERS-210 robots and resulted in a 20% increase in speed over our previous walking motion. When applied to the newly arrived ERS-7, it allowed us to reach speeds of up to 40 cm/sec. Details of the algorithm can be found in [3]. We believe our algorithm has many strengths over alternate methods. Even when starting from a random population, the algorithm is able to match the best previously known gait within a matter of hours. The GA-based approach is resistant to noise and avoids converging to local extremes.

6 Modeling the Effects of Kicking Motions

CMPack'03 had a wide variety of kicking motions. The addition of a new robotic platform has only added to the number of motions that the robot can use to manipulate the ball. We address the problem of modeling the effects of these different kicks. We then incorporate these models in the behaviors to select the most promising kick in a given state of the world [4]. As examples, we will discuss how the robot learned models of the trajectory angle and distance traveled in the forward and head kicks.

6.1 Trajectory Angle

The angle of the ball's trajectory after a kick is an important characteristic of all kicking motions. No external cameras or sensors are used in this experiment in order to make it applicable in any environment. The robot's vision module provides information about the ball's location for every vision frame where the ball is in the visual range of the camera. The ball position history records the position of the ball relative to the robot's frame of reference over the past 25 frames, or approximately 1 second. The ball history reports the ball position as unknown for frames where the ball is outside the camera field of view. Ball location estimates from the world model are not used in order to minimize error. Table 2 shows the algorithm we developed for approximating the angle of the ball's trajectory.

Algorithm 6.1: TRACKANGLE()

```
timeOfKick ← 0
do {
    TRACKBALLWITHHEAD()
    if BallWithinKickingRange = true
      then KICK()
           timeOfKick ← currentTime
    if currentTime − timeOfKick > t_delay
      then angle ← CALCANGLEUSINGBALLHISTORY()
           output (angle)
}
```

Table 2. Computation of the angle of ball's trajectory from an input of the estimated ball distance and angle values from vision.

The robot performs all analysis autonomously, and requires human assistance only in placing the ball in front of the robot for the next kick. The robot does not approach the ball on its own in order to assure the consistency of the trials.

The value of the variable t_{delay} is a key factor in the success of this procedure. Since the vision ball history is continuously keeping track of the ball's location, it is important to analyze the data in the history at the right time - before the ball is so far away that the values become unreliable, but after enough time has passed to record at least a second of the ball's trajectory. The standard value of t_{delay} is $t_{kick} + 1.0sec$, where t_{kick} is the duration of the kick. Slightly larger t_{delay} values are used for kicks with a lot of head movement that may cause a delay in tracking the ball after the completion of the kick.

A linear regression algorithm is applied to the points in the history buffer to calculate the angle of the ball's trajectory. The slope of the line is used to approximate the angle of the trajectory. In order to assure accuracy, we require that a minimum of 20 out of the last 25 vision frames contain information about the ball. Trials with fewer data frames are often a sign of an error. This can occur if the t_{delay} value is too small, if the kick fails or if the robot for some reason fails to locate and track the ball. Eliminating these trials ensures that inaccurate data is not introduced into the analysis and makes this learning process more efficient, requiring little human intervention.

6.2 Modeling Kick Distance

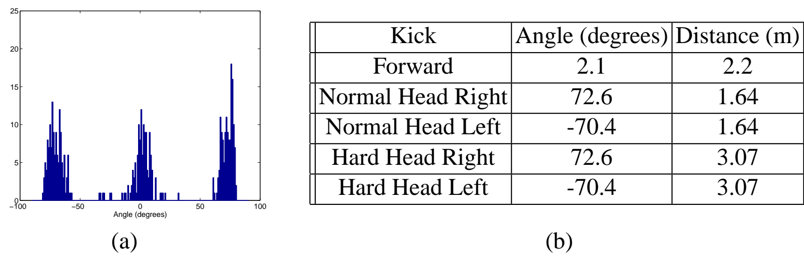
The second attribute important in understanding the effects of different kick motions is the distance the ball travels or the strength of the kick. The robot is unable to track the entire trajectory of the ball because the ball travels out of the robot's visual range for many of the powerful kicks. Instead, the robot calculates the final position where the ball stops relative to the original position of the robot before the kick. Table 3 shows the algorithm we developed for accurately calculating the displacement of the ball after a kick.

Algorithm 6.2: TRACKDISTANCE()

```
while 1
  do { APPROACHBALL()
       KICKBALL()
       STANDANDLOCALIZE()
       initialBallPosition ← currentRobotPosition
       FINDBALL()
       APPROACHBALL()
       if ballDistance < 50cm
         then { STANDANDLOCALIZE()
                finalBallPosition ← currentBallPosition
                ballDispVector ← finalBallPosition − initialBallPosition
                output (ballDispVector)
```

Table 3. Computation of a vector representing the ball's displacement relative to the location of the kick, given the estimates of the ball and robot locations from vision.

We selected two specific attributes to model the effects of the kicking motions, the angle of the ball's trajectory and the distance reached by the ball after the kick. Analysis of both kick attributes was performed using only the robot platform with no additional sensors and minimal human intervention. We used the acquired data to build a model that represents each kick in terms of its effects on the ball. Figure 4 (a) shows the angle variation for three kicks and Figure 4 (b) shows the average angle and distance reached by the ball for the different kicks. We did not gather experimental results for the hard head kick but we approximate the angle values to the ones of the normal head kick given their similarity with respect to angle.



Kick	Angle (degrees)	Distance (m)
Forward	2.1	2.2
Normal Head Right	72.6	1.64
Normal Head Left	-70.4	1.64
Hard Head Right	72.6	3.07
Hard Head Left	-70.4	3.07

(a) (b)

Fig. 4. (a) Ball angle distribution for a left head, forward, and right head kicks; (b) Average values for ball angle and ball distance for five kicks.

6.3 Selection of kicks

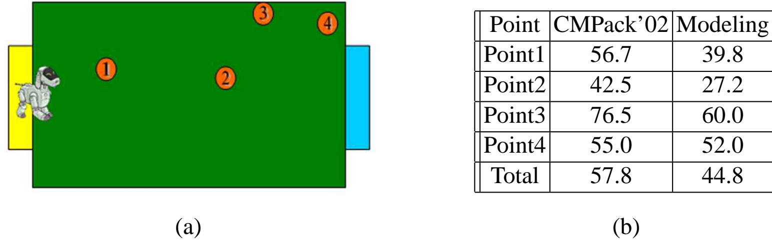
Once we have modeled the kick motions, we developed an algorithm for selecting an appropriate kick to use. The data gathered in the modeling stage can be organized into a motion library where each kicking motion is classified by its effects on the ball, mainly in terms of the mean angle and distance values. The library provides an easy reference and allows the user to easily add or remove motions.

The kick selection algorithm was designed to take advantage of this organization in the following manner. The localization system provides information about the robot's position on the field and the relative location of the goal, which allows the robot to calculate the desired trajectory of the ball. The motion library is then referenced to select the most appropriate kick whose effects match the desired trajectory. If no kick motion provides a close match, then a series of behaviors can be sequenced together to achieve the desired effect. For example the robot may choose to turn or dribble the ball in order to achieve a better scoring position. We performed several experiments to test the effectiveness of the selection algorithm.

Experimental results We report on an experiment to test the robot's ability to score a goal on an empty field without any other robots present. Testing single robot performance is important because it allows us to fine tune individual performance and analyze the robot's strategy without interference from other robots. Additionally, since the robot's vision of other robots is often inaccurate, this is one of the best metrics of performance.

Figure 5 (a) shows the four ball positions for the single robot scoring experiment. The robot is placed on the goal line of the defending goal. The location of the ball is

initially unknown to the robot and the ball may or may not be within visible range. The robot's performance is evaluated by recording the time it takes to score. The specific location of the four points for the position of the ball were selected to test various cases.



Point	CMPack'02	Modeling
Point1	56.7	39.8
Point2	42.5	27.2
Point3	76.5	60.0
Point4	55.0	52.0
Total	57.8	44.8

(a) (b)

Fig. 5. (a) Points 1-4 of the ball positions for robot scoring experiment with model-based kick selection. (b) Performance comparison of a CMPack'02 attacker and a CMPACK'04 attacker using the kick modeling selection algorithms. Values represent mean time to score in seconds, averaged over 13 trials per point.

– Point 1 tests the robot's ability to score from a large distance. The ball is located close to the initial position and in many cases the robot will require more than one kick to score.
– Point 2 tests the accuracy of the kicks. The point is located at a short distance and a direct angle to the goal. The robot is expected to score directly.
– Point 3 tests the robot's ability to score from the side when the ball is near a wall. The robot may or may not see the ball at this distance and often requires more than one kick to score.
– Point 4 tests situations when the ball is located in the corner at a wide angle to the goal. The robot does not see the ball at this distance. Scoring with one kick is very unlikely from this position.

The table in Figure 5 (b) summarizes the results of the experiment. A total of 104 trials were tested, with 13 trials for each ball position. Results show that the modeling and prediction algorithm performed better for every point, with an overall improvement of 13 seconds. The statistical significance of the results was confirmed using the Wilcoxon Signed Rank Test.

The new kicking selection algorithm showed to be effective in game situations in which the robots use the model of the kicks for selecting the most promising kick. Our modeling research is ongoing and we envision experimentally gathering and using new models of the effects of the robot's actions. We are especially interested in devising algorithms to improve the action selection of the robots in the presence of complex multi-robot situations.

7 Supporter Positioning

We support two different modes of supporter positioning. One of these is based on potential fields; the other we call ”constraint-based” positioning. The potential field-based method is equivalent to the potential-field based positioning we have used in previous years, though some constants have been changed. The constraint-based method was what we used at the RoboCup competition.

In constraint-based positioning, we select a target point that the supporting robot should walk to, and it attempts to walk straight there while facing the ball. For the defender, this point is midway between the ball and the center of the defense goal. For the supporting attacker, this point is generally on the opposite side of the primary attacker and further ”upfield”, though there are specific cases where different positions are chosen. These ”normal” positions can be overridden by certain higher-priority constraints, such as the constraint that the robot not enter the defense box, or stand within a certain distance of its teammates.

8 Conclusion

With CMPACK'04 , we pursue our research on teams of intelligent robots. We continue to note that a team of robots needs to be built of skilled individual robots that can coordinate as a team in the presence of opponents. In this paper, we focused on presenting our high-level vision for constructing a local model, the local model itself, a novel method of state detection from accelerometer data, and empirical modeling of kicking motions.

Our work for next year will continue to focus on both individual and team skills as well as examine ways to model the world in a principled fashion.

Acknowledgements

This research has been partly supported by Grant No. DABT63-99-1-0013, by generous support from Sony, Inc., and by a National Physical Science Consortium Fellowship with stipend support from HRL Laboratories. The content of this publication does not necessarily reflect the position of the funding agencies and no official endorsement should be inferred.

References

1. J. Bruce, T. Balch, and M. Veloso. CMVision, www.cs.cmu.edu/~jbruce/cmvision/.
2. J. Bruce, T. Balch, and M. Veloso. Fast and inexpensive color image segmentation for interactive robots. In *Proceedings of IROS-2000*, 2000.
3. Sonia Chernova and Manuela Veloso. An evolutionary approach to gait learning for four-legged robots. In *Proceedings of IROS 2004*, 2004.
4. Sonia Chernova and Manuela Veloso. Learning and using models of kicking motions for legged robots. In *Proceedings of ICRA 2004*, 2004.
5. S. Lenser, J. Bruce, and M. Veloso. CMPack: A complete software system for autonomous legged soccer robots. In *Autonomous Agents*, 2001.

6. J. R. Quinlan. *C4.5 – Programs for Machine Learning*. The Morgan Kaufmann series in machine learning. Morgan Kaufman Publishers, 1993.
7. William Uther, Scott Lenser, James Bruce, Martin Hock, and Manuela Veloso. CM-Pack'01: Fast legged robot walking, robust localization, and team behaviors. In A. Birk, S. Coradeschi, and S. Tadokoro, editors, *RoboCup-2001: The Fifth RoboCup Competitions and Conferences*. Springer Verlag, Berlin, 2002.
8. Douglas Vail and Manuela Veloso. Learning from accelerometer data on a legged robot. In *Proceedings of IAV 2004*, 2004.
9. M. Veloso and W. Uther. The CMTrio-98 Sony legged robot team. In M. Asada and H. Kitano, editors, *RoboCup-98: Robot Soccer World Cup II*, pages 491–497. Springer, 1999.
10. M. Veloso, E. Winner, S. Lenser, J. Bruce, and T. Balch. Vision-servoed localization and behavior-based planning for an autonomous quadruped legged robot. In *Proceeding of the 2000 International Conference on Artificial Intelligence Planning Systems (AIPS'00)*, 2000.