Cerberus'08 Team Report

H. Levent Akın[1]
Çetin Meriçli[1]
Tekin Meriçli[1]
Barış Gökçe[1]
Ergin Özkucur[1]
Can Kavaklıoğlu[1]
Olcay Taner Yıldız[2]

[1]Artificial Intelligence Laboratory
Department of Computer Engineering
Boğaziçi University
34342 Bebek, İstanbul, Turkey

[2]Department of Computer Engineering
Işık University
Sile, İstanbul, Turkey
{akin, cetin.mericli, tekin.mericli}@boun.edu.tr
{sozbilir, nezih.ozkucur, can.kavaklioglu}@boun.edu.tr
olcaytaner@isikun.edu.tr

November 1, 2008

Contents

1

1 Introduction **2**

2 Software Architecture **3**
2.1 BOUNLib . . . 3
2.2 Cerberus Station . . . 3
2.3 Cerberus Player . . . 4
2.3.1 Common library . . . 4
2.3.2 Vision library . . . 5
2.3.3 Planner library . . . 5
2.3.4 Robot specific elements . . . 5
2.3.5 Cerberus Player Module on Nao . . . 5

3 Vision **10**
3.1 Image Processing and Perception . . . 10
3.1.1 Color Classification . . . 10
3.2 Scanline Based Perception Framework . . . 12
3.2.1 Vision Object . . . 12
3.2.2 Goal Perception . . . 12
3.2.3 Line Perception . . . 16
3.2.4 Goal Target Perception . . . 17
3.2.5 "On Goal Line" Perception . . . 17
3.2.6 Distance Calculations . . . 18

4 Localization **19**
4.1 World Modeling and Short Term Observation Memory . . . 20
4.2 Localization . . . 21

5 Motion **25**
5.1 Motion Engine . . . 25
5.2 Implemented Biped Walking Algorithms . . . 26
5.2.1 Static Walking with CoM Criterion . . . 27
5.2.2 Dynamic Walking with CPG Method . . . 27

6 Planning and Behaviors **37**

Chapter 1

Introduction

The **Cerberus** team made its debut in RoboCup 2001 competition. This was the first international team participating in the league as a result of the joint research effort of a group of students and their professors from Boğaziçi University (BU), Istanbul, Turkey and Technical University Sofia, Plovdiv branch (TUSP), Plovdiv, Bulgaria. The team competed in Robocup 2001-2008 except the year 2004. Currently Boğaziçi University is maintaining the team. In 2005, despite the fact that it was the only team competing with ERS-210s (not ERS210As), Cerberus won the first place in the technical challenges. In the Robocup 2008 organization [1], a new league, named the Standard Platform League (SPL) [2], was introduced. In this league, the Nao humanoid robots manufactured by Aldebaran Robotics [3] are used as the standard robot platform and no hardware modifications are allowed as was the case for the 4-Legged League with Aibo robots. Cerberus competed in both the 4-legged and the 2-legged categories of the SPL in 2008 and made it to the quarterfinals, losing to the eventual champion in the 4-legged category.

The organization of the rest of the report is as follows. The software architecture is described in Chapter 2. In Chapter 3, the algorithms behind the vision module are explained. The main localization algorithm is given in Chapter 4. The planning module is described in Chapter 6. The locomotion module and gait optimization methods used are explained in Chapter 5.

Chapter 2

Software Architecture

Software architecture of Cerberus has been completely rewritten in 2008. The existing modular architecture was transformed into a more general library architecture, where the code repository is separated into levels in terms of generality. Similar to the well known Model-View-Control architecture, the main goal of this new approach was to organize our code base into logical sections all of which are easy to access, manipulate and debug. The rewrite process was originally targeting the Aibo platform but the well designed architecture has made our initial development on Nao painless and quick.

Software architecture of Cerberus consists of mainly three parts:

- BOUNLib
- Cerberus Player
- Cerberus Station

2.1 BOUNLib

Past experience has demonstrated the previous modular approach to be sub-optimal in some cases. Especially considering issues such as reuse of source code for multiple architectures and also multiple purposes, making specific modifications to the special purpose modules becomes very time consuming and error prone.

We have started collecting general parts of our code base in a library structure called *BOUNLib*. Using this library will enable us to easily code for different platforms or different robots by reusing most of our code base.

2.2 Cerberus Station

BOUNLib library includes a versatile input output interface, called *BOUNio*, providing essential connectivity services to the higher level processes such

as reliable UDP protocol, file logging, and TCP connections. Connections are made seamlessly to the sender, thus there is no need to write specific code for any application or test case.

Using *BOUNio* library enabled us to implement a very general version of our previous *Cerberus Station* using Trolltech's Qt Development Framework [4]. Using the well structured architecture of our runtime code and Cerberus Station, it is very easy to test new features to be added to the robot, which is a very vital resource for any research experiment.

Cerberus Station is designed to have the same features of old *Cerberus Station* and more, mainly aimed at visualizing the new library based code repository, some of which are listed below:

1. Record and replay facilities providing an easy to use test bed for our test case implementations without deploying the code on the robot for each run.
2. A set of monitors which enable visualizing several phases of image processing, localization, and locomotion information.
3. Recording live images, classified images, intermediate output of several vision phases, objects perceived and estimated pose on the field in real time.
4. Log to file and replay at different speeds or frame by frame.
5. Locomotion test unit in which all parameters of the motion engine and special actions can be specified and tested remotely.

2.3 Cerberus Player

Following the same design pattern, *Cerberus Player* code base was also switched to a more robust library structure instead of the previous static modular approach. The new design consists of four main elements:

- Common library
- Vision library
- Planner library
- Robot specific elements

2.3.1 Common library

The *Common library* is dedicated to containing robot data and their primitive functions such as serialization/deserialization. The primary element of the *Common library* is the *Robot* class, which defines the data elements a

robot needs to keep track of in the run time. The *Robot* class can be considered as the state vector of the robot at a time. Using this single source of data greatly simplifies some of the architectural constraints. Other elements of the *Common library* include, sensor related, vision related classes and configuration related classes.

2.3.2 Vision library

The *Vision library* is designed to accommodate multiple vision algorithms seamlessly. Using such a standard interface provides a great visual research platform, where any new algorithm can be tested easily and thoroughly. Using our new approach, it will be possible to test our existing vision algorithms against the new scanline alternatives, providing us a very fruitful test environment to compare multiple techniques and reach precise conclusions with thorough analysis.

2.3.3 Planner library

The *Planner library* is refactored with the new architecture. Having a *Planner library* enables us to use the implemented planning facilities on other platforms as well.

2.3.4 Robot specific elements

Any general code requires a point of connection to a specific architecture. Robot specific elements provide the bindings between the library components and the robot hardware.

2.3.5 Cerberus Player Module on Nao

There are two ways for a controller to use (or communicate with) *NaoQi* on the Nao robot.

- The first way is to develop a different executable other than *MainBroker* which runs as a separate process and communicates with *MainBroker* over network (Figure 2.1).
- The second way is to develop a new module and loading it along with the *MainBroker*. This way, function calls are faster (Figure 2.2).

The user modules which are loaded with *MainBroker* are less safe than a separate process. If an exception occurs, all the hardware controller would crash and cause hardware damage. However, this method is faster than separate process.

A soccer playing system is very complicated and there is always a high possibility of getting run time exceptions especially during development.

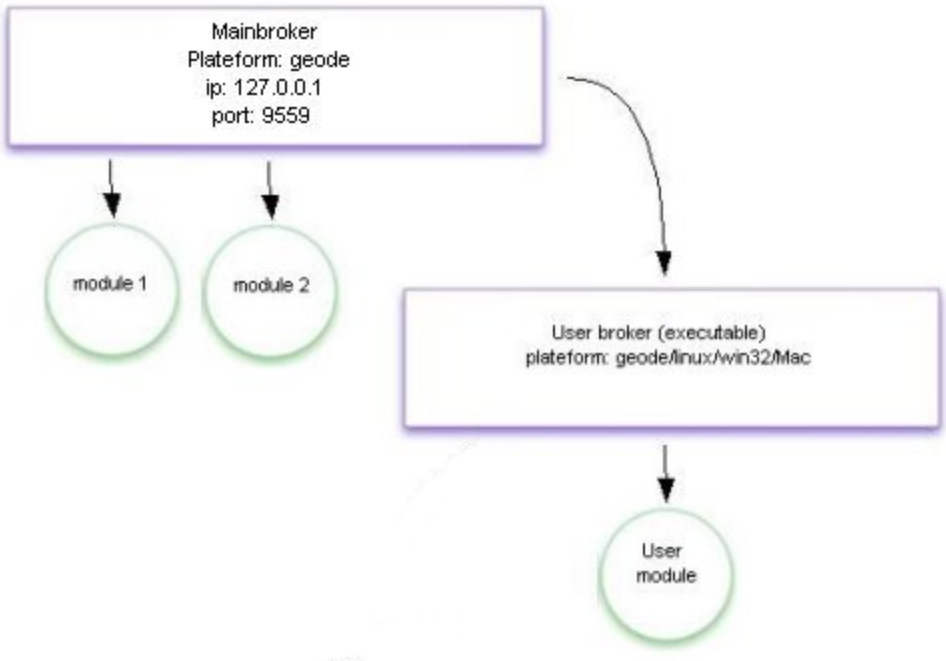


Figure 2.1: Safe user controller and *MainBroker*.

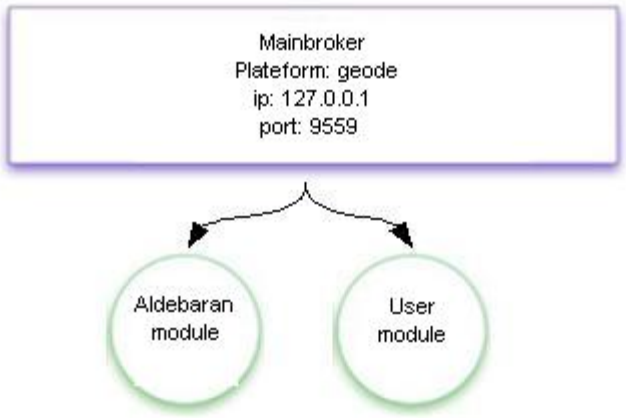


Figure 2.2: Unsafe but faster user controller and *MainBroker*.

On the other hand, such a system should be as fast as possible. After considering the circumstances, we developed our system to support both execution methods. The separate process method is used frequently in early development of system components, and module loading method is used for testings in real time.

Our soccer playing system is basically a module which uses other modules. Except actual soccer related components, this module has two main jobs for hardware management; image synchronization and the motion management.

Image Synchronization Process

Current Nao platform has one camera. The camera can provide 5, 15 or 30 frames in one seconds optionally. The raw format of the image is YUV422. In the YUV422 format, the colors are represented with three channels as Y U and V. Each consecutive two pixels share same U and V values. In Figure 2.3, the memory layout of the image format is given.

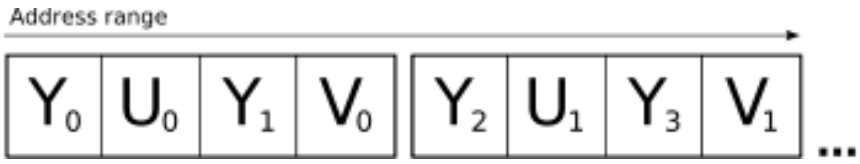


Figure 2.3: Memory layout of the YUV422 image format.

To take images from the camera, first a *proxy* is created and connected to *ALVision* module. Before taking images, camera settings are set, which are in our case 15 frames per second, YUV422 image format, 320x240 resolution and auto white balance disabled. After setting the parameters, a new thread is created which will constantly request an image from the *ALVision* module and process all related works. After process is finished, instead of immediate request of the new image, the thread is put to sleep state. The sleep time is estimated as in the algorithm given in Figure 2.4.

Motion Manager Process

Motion manager process is a separate thread and is responsible for constantly sending the most recent servo commands to the *ALMotion* module.

Synchronization of Image and Motion Manager Process

There is a constant data exchange between the motion management process and the image fetching process. The planning algorithm (explained in Chapter 6) sends motion commands to the motion manager and in turn it sends the odometry values to the localization algorithm. Note that both the

```
1: NewImage ← 0
2: PreviousImage ← 0
3: while true do
4:    NewImage ← FetchImage()
5:    while NewImage = PreviousImage do
6:       sleep 2 ms
7:       NewImage ← FetchImage()
8:    end while
9:    b ← time()
10:   Process Image
11:   elapsedTime ← time() − b
12:   if elapsedTime < 1000/15 then
13:      sleep for 1000/15 − elapsedTime ms
14:   end if
15: end while
```

Figure 2.4: Image fetching algorithm.

planning and localization algorithms are executed by the image processing thread right after the perceptual algorithms' termination.

As explained in 2.1, all the data are represented in *BOUNCore* class which is created uniquely for all the system. The odometry and motion commands are included in the *BOUNCore* class and synchronization between the two threads must be done. In Figure 2.5, the synchronization between two threads is shown.

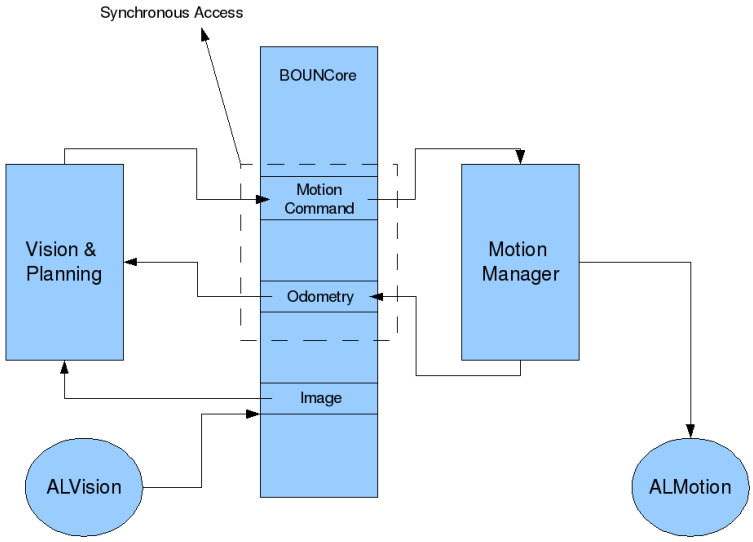


Figure 2.5: Synchronization between image process thread and motion manager thread.

Chapter 3

Vision

3.1 Image Processing and Perception

The purpose of the perception module is to process the raw image and extract available object features from the image (bearing and range if available). The input to the module is the image in YUV422 format as explained in 2.3.5 and the output is the range and bearing features of seen objects on the field. The two main parts of this module are color classification and perception.

3.1.1 Color Classification

In the raw image format, each pixel is represented with a three-byte value and can be one of the 255^3 values. Since it is impossible to efficiently operate on such an input space, the colors are classified into fewer color groups. Each group represents a possible color on the environment. These are white, green, yellow, blue, robot-blue, orange, and red. There is actually one more group which stands for none of the groups and noted as the "ignore" group.

To summarize the problem, we have an image with a resolution of $320x240$, where each pixel has a range of 255^3 values and we want to assign each pixel to one of the eight color groups. This is a *regression* problem and we used Generalized Regression Neural Network (GRNN) to solve this problem [5]. GRNN is a neural network which can approximate a function (linear or nonlinear) with n dimensional domain to m dimensional output domain.

In our problem, input space has three dimensions and output space has eight dimensions. In the output, the value with index of the target color group has the value of one while all other values have value of zero. To reduce the complexity of the algorithm, we omitted two bits from Y channels and four bits from U and V channels. The new input space is still three dimensional but has a range of $2 \times 6 \times 2^4 * 2^4$.

Before training a GRNN network, a training set must be formed. For this purpose, we use our *Labeler* software. In the *Labeler*, an image in its raw format is converted to RGB format and displayed. The user marks sample pixels with their color group. When sufficient number of samples are collected from different images, it is saved as the training set. In Figure 3.1, a screen shot from *Labeler* software is shown.

Figure 3.1: Screenshot from *Labeler* software. Grey region is the ignored color group.

After the ceation of the training set, GRNN is trained. To efficiently get outputs from the GRNN network, a look up table is constructed for all possible inputs. To look up the color group of a pixel, Y, U and V values are used to calculate the unique index and the value at that index gives the color group ID. In Figure 3.2, output of a trained GRNN is tested on a sample image.

Figure 3.2: A classified image constructed with a trained GRNN.

3.2 Scanline Based Perception Framework

Scanline based vision framework was introduced this year to exploit several advantages of scanline based perception systems over blob based vision systems.

The most significant improvement over blob based systems is the increased reasoning possibilities. Using a single scanline it is possible to detect a line segment by simply tracking the change in color of the pixels on the scanline. Furthermore, it is possible to combine information gained by multiple scanlines providing further spatial information about the structures available in the current image. In our previous blob based perception system such extensive and precise reasoning was not possible.

Scanline perception framework is also more scalable. The complexity of the system can be changed automatically in run time, by adjusting the number of scanlines per image. Since only the pixels on the scanlines are segmented, segmentation overhead of the blob based systems can also be avoided.

Inspecting a region of the image with a set of scanlines can provide a fast and accurate method of reasoning about the confidence of the perception. Such confidence values are very crucial for the performances of higher level modules. Further probabilistic methods will be employed to increase utility of these confidence values in the future.

In the following parts of this section, the implemented scanline based perception methods will be described in detail.

3.2.1 Vision Object

The *Vision* object instantiates other perception methods after generating some information which is required by all of the specialized perception classes.

The image received from robot's camera is first scanned with several scanlines. The number of these scanlines can be altered according to the viewing angle of the camera, however currently this feature is left as future work.

Once pixels corresponding to the scanlines are segmented, this segmented and ordered set of pixels are traversed once for important points, defined by each specialized perception method. Selected subsets of the segmented pixels are then sent to specialized perception classes for further processing and object detection.

3.2.2 Goal Perception

This section describes the most extensive specialized perception class, which is designed for goal perception. Processing starts with the input of the vision

object, indicating the important points for this specialized perception class.

The goal of the scanline based goal perception is to detect the goal bars individually so that multiple landmarks can be perceived from a single goal. A four stage procedure is designed to achieve this purpose, which can be used to perceive left and right goal bars individually. Due to the generic design of the perception stages and the underlying scanline framework, implementing top and bottom bars is only a task of mirroring left and right bar perceptors. Similarly beacon perception will be only a variation of the bar perception.

Tests of the perceptor are done using the new version of the *Cerberus Station*, a screen shot of which is shown Figure 3.2.2. Using this visual tool greatly enhances the testing procedure. Employing the *BOUNio* library it is possible to observe run time performance as well as to inspect recorded logs within the goal perception tester tool.

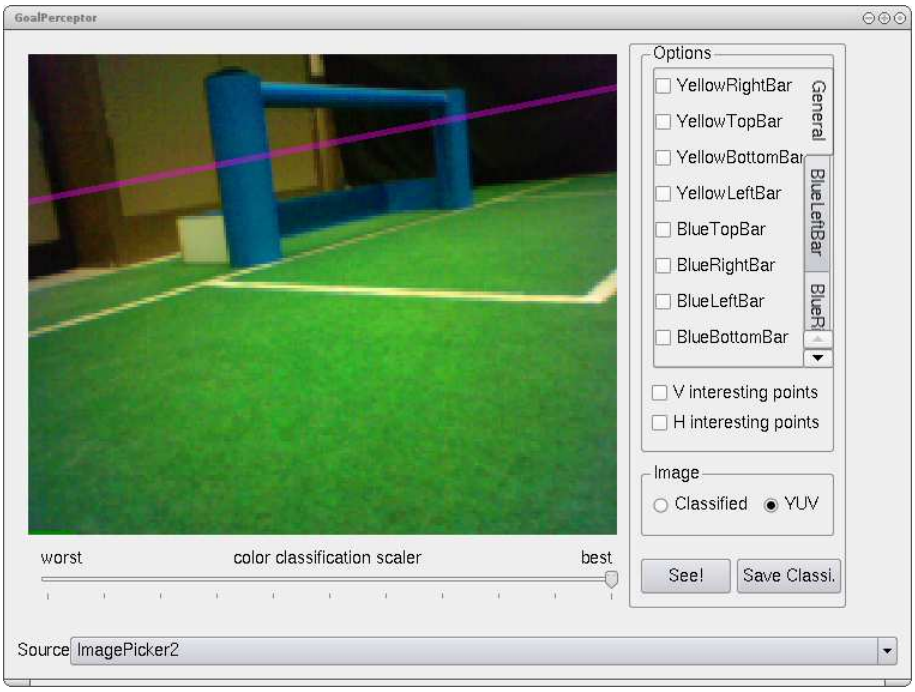


Figure 3.3: Cerberus Station Goal Perception Test Tool.

The goal perception task is divided between several classes, namely main *vision* object, goal perceptor, goal bar perceptor, and bar perceptor. The bar perceptor performs the four stage bar perception procedure, the goal bar perceptor investigates if the found bar is the correct bar, and the main vision object initiates the procedure.

Initially the main *vision* class runs a series of scanlines perpendicular to the horizon plane and classifies the YUV image along these scanlines (see

Figure 3.2.2). Using this procedure it is possible to avoid classifying all of the image, which accounts to a large portion of the computational cost in previous implementations.

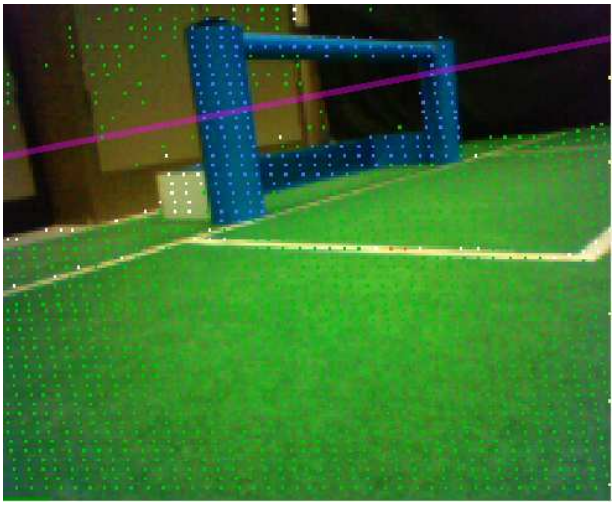


Figure 3.4: Important points.

Generated classified points, called *important points* are passed to the goal perceptor class. There are four goal bar perceptor classes, one for each vertical bar of the two goals, blue left, blue right, yellow left, yellow right.

Each of these goal bar perceptors have three bar perceptors. Each goal perceptor inspects all of these three bar perceptions to pick the best candidate to be chosen as the perceived goal bar.

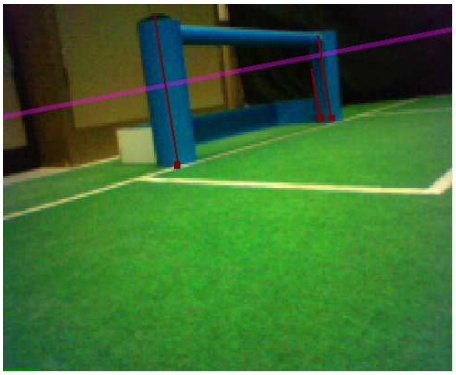
Finally the bar perceptor follows a four stage procedure to detect the goal bar candidates as follows (see Figure 3.2.2):

1. **Find vertical base:** The *important points* are scanned for the largest continuous segment of the given color (blue or yellow in this case). The segment can be discontinued with a value defined by a constant, which can be adjusted according to the performance of the available classification .

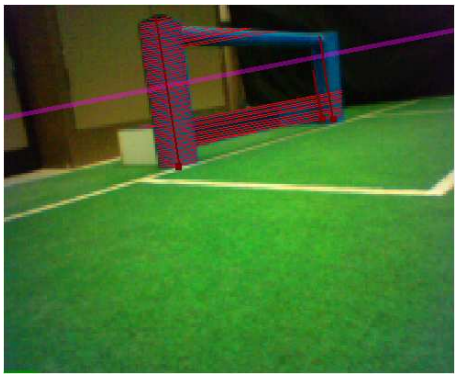
2. **Ranger lines:** From the detected vertical base, perpendicular scanlines are drawn to locate the ends of the colored region.

3. **Bound detection:** Ranger lines are passed through a histogram function to find the most likely edge of the bar.

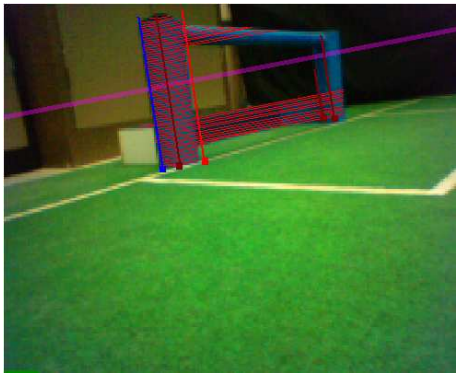
4. **Bounding box:** The four corners of the detected bounds are stored as the perceived bar plane.



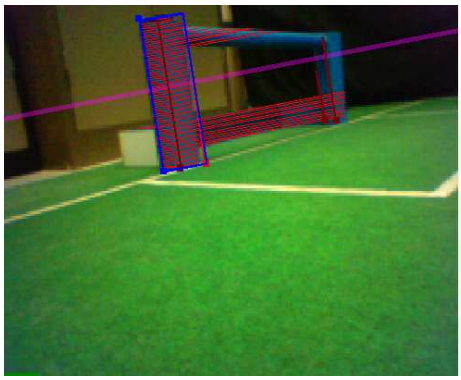
(a) Vertical bases.



(b) Ranger lines for the largest vertical base.



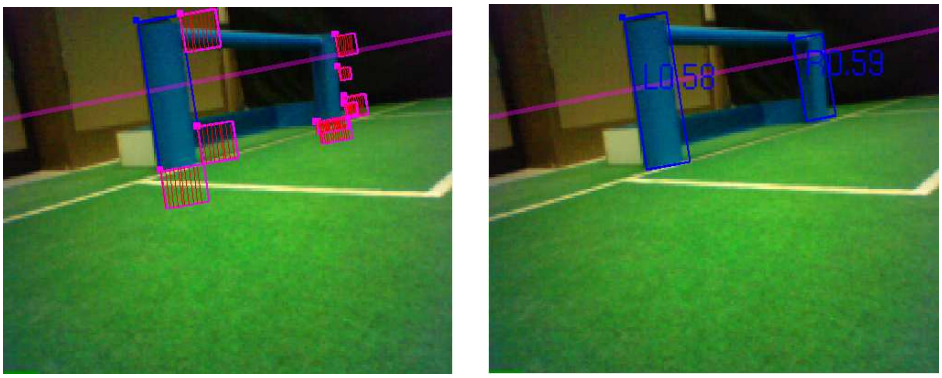
(c) Bounding lines.



(d) Final bounding box.

Figure 3.5: Four stages of bar perception

Once candidate bars are generated, respective goal bar perceptors search for the best available vertical goal bar of their respective colors. To achieve quantified confidence results, goal bar perceptors draw squares, scaled to the size of the perceived bar, around the expected positions of the colored regions, as shown in Figure 3.2.2a. Within these squares scanlines are run to detect the ratio of pixels with the requested color. Figure 3.2.2b shows the resulting confidence values. Observations indicate false positive frequency is very low with confidence values greater than 0.5.



(a) Confidence regions drawn for all possible left blue vertical goal bars.

(b) Final confidence values for left and right blue vertical goal bars.

Figure 3.6: Final perception.

3.2.3 Line Perception

Line perception is implemented in a similar fashion with the goal target perception. The main vision object supplies the *important points* for the line perception, which is, in this case, defined as the mid point of a green-white transition and a consecutive white-green transition on a single scanline.

First, the best possible fitting lines are found for the current *important points* with least squares line fitting. Using these lines a circle detection algorithm runs, trying to find the center circle of the field. In case the middle circle is found, no further processing is done. Since there are many intersection points on the center circle lines, proceeding further with corner detection gives false positive results. Besides, finding the center circle is very informative as it is.

In case the center circle is not found, corner detection algorithm is run to find the corners formed by the detected lines. After several sanity checks, the found corners are marked as the perceived corners.

3.2.4 Goal Target Perception

Using the available scanline perception framework it is quite easy to implement simple perceptions quickly having all the robust properties of the scanline vision technique. Goal target perception presents a good example for such a case.

In order to score a goal, a goal target perception is required, so that the robot can shoot at the correct spot. Localization can generally be relied upon to figure out the direction of the goal, however it can not be trusted for precise shooting due to high level of noise.

This perception method provides a more limited but more accurate approach to be used as a precise goal target on the perceived image, once the robot is assumed to be actually facing the goal. There are sanity checks in place to avoid shooting in case the robot does not actually face the goal.

The *important points* for this very specialized perception are the points colored with the opponent team's goal color. These points are traversed for the largest continuous region obeying the sanity checks. If no such region is found, goal target perception can set goal target to be *unknown* and avoid shooting to a completely obstructed goal or even worse to any other random direction. If the region is found as an appropriate goal target, its center is marked as the goal target.

3.2.5 "On Goal Line" Perception

The most specialized perception is "on goal line" perception which provides a signal to the goal keeper to help the robot with its localization. Once the goal keeper roughly gets around the goal line, it can be quite hard to get localized precisely. Since the opponent goal is quite far away and generally obstructed, it can rarely be used in localization, which degrades the performance of the localization.

To help the goal keeper in such situations, "on goal line" perception signals the goal keeper that it is standing on the goal line. The planner module of the goal keeper turns the robot's head from one side to the other all the time, searching for a possible ball to save. This behavior frequently results in a particular sequence of perceptions when the goal keeper is on the goal line, which are the own goal bar perception - corner perception - corner perception - own goal bar perception, respectively.

"On goal line" perception looks for such sequences and generates a signal indicating the goal keeper is on the goal line, which can be used to align the goal keeper more precisely to improve chances for a better goal saving and localization performance.

3.2.6 Distance Calculations

To calculate the distance of a visually observed object, either a stereo vision system or using consecutive frames of single camera as stereo vision is needed [6]. The Nao robot has a single camera but the sizes of the objects in the environment are fixed and known. The distances of objects can be inferred by the size feature of the objects. In visual observations, the size of an object can be measured by the width or height of the object in pixels. As the object gets closer, the size of the object in the image increases and this increase has a nonlinear relationship with the actual distance of the object. In our approach, we represent this relationship with the function $y = a \times (x^b) + c$ where y is the distance, x is the width or height of the object in pixels, and a, b and c are parameters of the function.

Now the problem is reduced to finding the three optimal values for a, b and c. To optimize these parameters, a training set is collected from the environment and the parameters are estimated with the least squares method. The pixel widths are found from the classified image, so this training procedure is repeated for each object type and after each color calibration process which was explained in section 3.1.1.

Chapter 4

Localization

Currently, Cerberus employs three different localization engines. The first engine is an inhouse developed localization module called Simple Localization (S-LOC) [7]. S-LOC is based on triangulation of the landmarks seen. Since it is unlikely to see more than two landmarks at a time in the current setup of the field, S-LOC keeps the history of the percepts seen and modifies the history according to the received odometry feedback. The perception update of the S-Loc depends on the perception of landmarks and the previous pose estimate. Even if the initial pose estimate is provided wrong, it acts as a kidnapping problem and is not a big problem as S-Loc will converge to the actual pose in a short period of time if enough perception could be made during this period.

The second one is a vision based Monte Carlo Localization with a set of practical extensions (X-MCL) [8]. The first extension to overcome these problems and compensate for the errors in sensor readings is using inter-percept distance as a similarity measure in addition to the distances and orientations of individual percepts (static objects with known world frame coordinates on the field). Another extension is to use the number of perceived objects to adjust confidences of particles. The calculated confidence is reduced when the number of perceived objects is small and increased when the number of percepts is high. Since the overall confidence of a particle is calculated as the multiplication of likelihoods of individual perceptions, this adjustment prevents a particle from being assigned with a smaller confidence value calculated from a cascade of highly confident perceptions where a single perception with lower confidence would have a higher confidence value. The third extension is related with the resampling phase. The number of particles in successor sample set is determined proportional to the last calculated confidence of the estimated pose. Finally, the window size in which the particles are spread into is inversely proportional to the confidence of estimated pose. This engine was used in both Aibo and Nao games.

The third engine is a novel contribution of our lab to the literature,

Reverse Monte Carlo Localization (R-MCL) [9]. The R-MCL method is a self-localization method for global localization of autonomous mobile agents in the robotic soccer domain, which proposes to overcome the uncertainty in the sensors, environment and the motion model. It is a hybrid method based on both Markov Localization (ML) and Monte Carlo Localization (MCL) where the ML module finds the region where the robot should be and MCL predicts the geometrical location with high precision by selecting samples in this region (Fig. 4.1). The method is very robust and requires less computational power and memory compared to similar approaches and is accurate enough for high level decision making which is vital for robot soccer. This method is still in testing and we are planning to integrate it in our system in 2009.

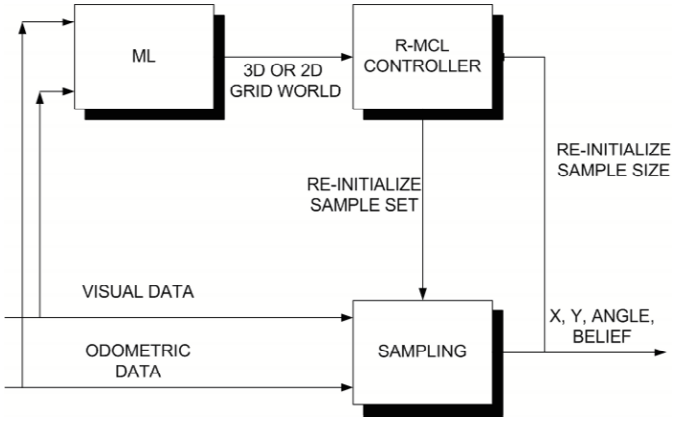


Figure 4.1: R-MCL Working Schema.

4.1 World Modeling and Short Term Observation Memory

The perception module provides instantaneous information. While reactive behaviors like tracking the ball by head requires only instant information, other higher level behaviors and localization module needs more.

The planning and localization modules, a perceptual information with less noise and more complete. The world modeling module should reduce sensor noise and complete the missing state information by predicting the state. This is a state prediction problem and we use the most common approach in the literature which is the Kalman Filter [10].

The Kalman filter assumes the noise in both prediction and update steps have Gaussian distribution. A linear state evolution and perceptual model is required to apply Kalman filter. If the models are non-linear Extended

Kalman Filter can be used where models are linearized by Taylor Series.

In our problem, the observations are distance and the bearing of the objects with respect to robot origin. And the state we want to know is actual distance and bearing informations. In addition for the dynamic objects like the ball, the state vector also includes distance change and bearing change information to ease prediction.

For any object, the observation is $z = \{d, \theta\}$ where d and θ are distance and bearing respectively to the robot origin. For the stationary objects state is $m = \{d, \theta\}$ and the state evolution model is $m^1_{k+1} = I * m_k$ and $z_k = I * m_k$ where k is time and I is the unit matrix.

For the dynamic objects, the observation is the same but state is represented as $m = \{d, \theta, d_d, d_\theta\}$ where d_d is the change in distance in one time step and d_θ is the change in bearing likewise. The state evolution model is:

$$\begin{pmatrix} d_{k+1} \\ \theta_{k+1} \\ d_{d,k+1} \\ d_{\theta,k+1} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} d_k \\ \theta_k \\ d_{d,k} \\ d_{\theta,k} \end{pmatrix}$$

and observation model is:

$$\begin{pmatrix} d_{k+1} \\ \theta_{k+1} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} d_k \\ \theta_k \\ d_{d,k} \\ d_{\theta,k} \end{pmatrix}$$

As can be observed from the model specifications, we omit the correlation between objects and separately execute filter equations for each object. If an object is not observed for more than a pre-specified time step, the belief state is reset and the object is reported as unknown. For our case this time step is 270 frames for stationary objects and 90 frames for dynamic objects.

In the update steps, the odometry readings are used. The odometry reading is $u = \{d_x, d_y, d_\theta\}$ where d_x and d_y are displacements in egocentric coordinate frame and d_θ is the change in orientation. When an odometry reading is received, all the state vectors of known objects are geometrically re calculated and the uncertainty is increased.

The most clear effect of using a Kalman Filter is that the disadvantage of limited field of view is reduced. In the Figure 4.2, a robot spins its head and is aware of three distinct landmarks at the same time.

4.2 Localization

In the previous years in SPL, a simple goal box seeking action was enough to score or even win matches. However, in recent years, the score intented actions of the robot are highly dependent on self pose information. The

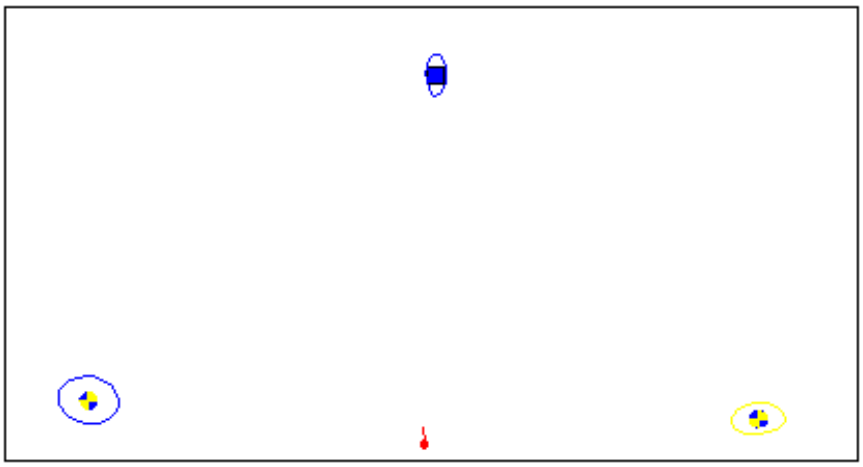


Figure 4.2: A robot spins its head and is aware of three distinct landmarks at the same time.

problem of estimating the self pose has also become harder due to the decreased number of landmarks and increased field size.

The localization problem is a self pose estimation problem like in section 4.1, and the widely used approach is the Monte Carlo Localization (MCL) algorithm [11]. In this problem, the state to be estimated is $\mu = \{x, y, \theta\}$, where x and y are global coordinates and θ is orientation of the robot.

In MCL algorithm, belief state is represented by a particle set and each element represents a possible pose of the robot. In Figure 4.3, a sample belief state is given.

In the MCL algorithm, an importance weight is calculated for each particle which shows how suitable a particle to the observations. All the particles are copied to the next belief state set with respect to their importance weight. To calculate importance weights, the observations are taken from the output of the world modeling algorithm as described in section 4.1. Only the known stationary objects are used in calculations. The importance weights is calculated by difference of expected bearing and predicted bearings and likely if the distance is known, expected distance and predicted distance. In Figure 4.4, an example belief state is given, where both distance and bearing is known for a single landmark.

While importance weight and resampling steps are the correction step of the MCL, prediction step is done with odometry readings. In prediction, each particle is updated with geometrically recalculating the position and adding a small random noise which represents increase in uncertainty. To avoid kidnapping situations, in the resampling step, some of the particles are randomly generated.sampling steps are the correction step of the MCL, prediction step is done with odometry readings. In prediction, each particle is updated with geometrically recalculating the position and adding a small

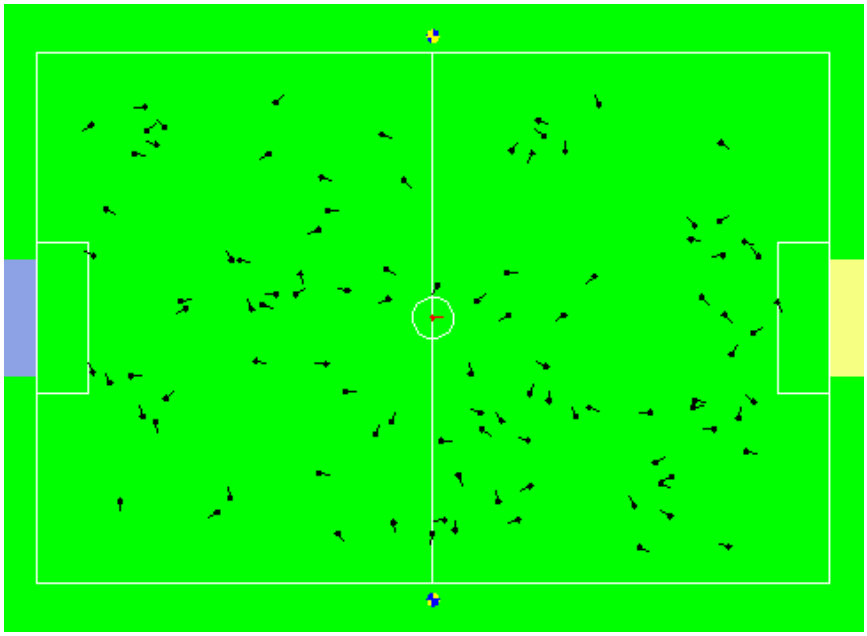


Figure 4.3: Belief state of the robot in MCL algorithm.

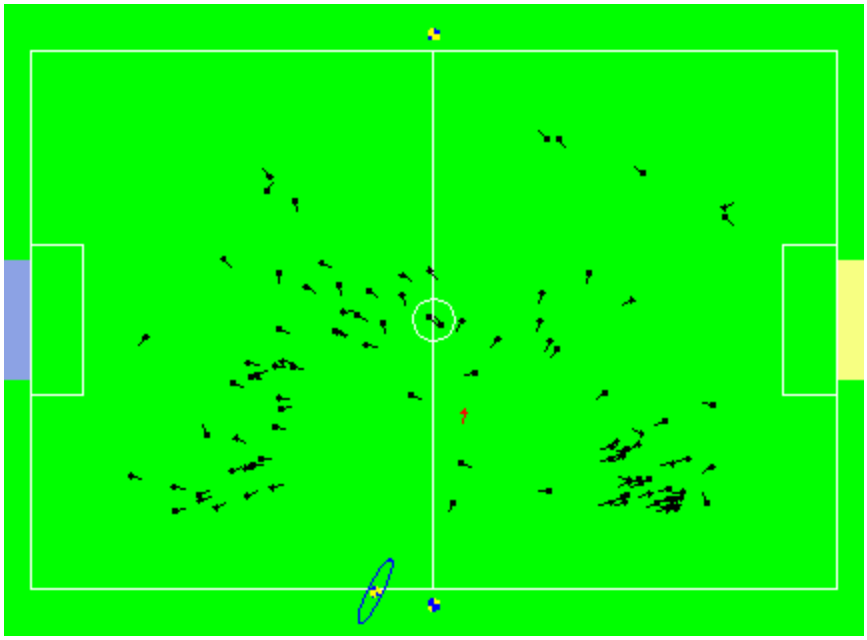


Figure 4.4: An example belief state is given, where both distance and bearing is known for a single landmark. The particles are grouped over an arc around true position of the object.

random noise which represents increase in uncertainty. To avoid kidnapping situations, in the resampling step, some of the particles are randomly generated.

When we need the best estimate of the pose, we take the mean of a subset of particles with highest importance weights. In Figure 4.5, the best estimate of the robot with belief state is given.

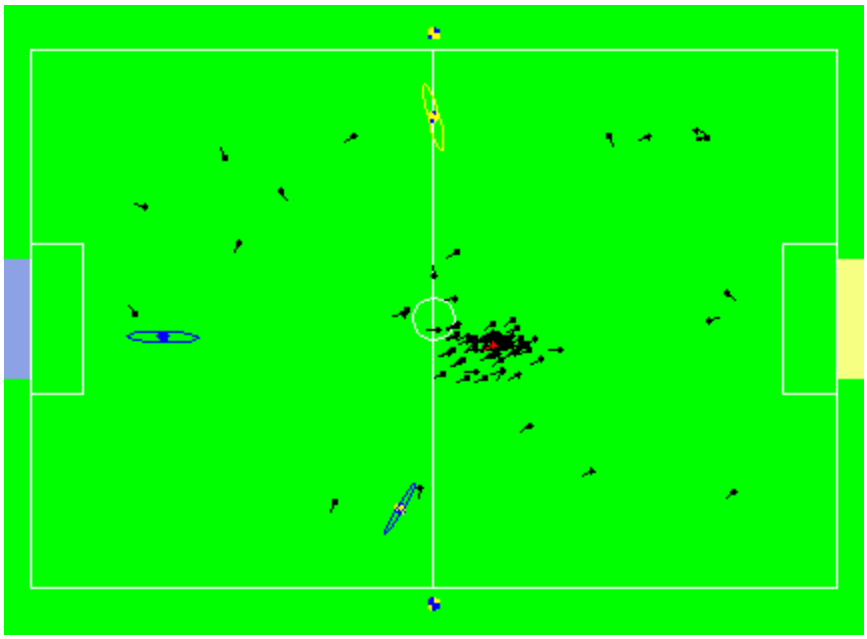


Figure 4.5: The best pose estimate is marked in red color.

Chapter 5

Motion

For our team, the most important component of transition from quadruped robots to biped robots was the locomotion module. Fast and robust locomotion is a vital capability that a mobile robot should have, especially if the robot is expected to play soccer. Cerberus has a significant research background in different mobility configurations including wheeled and legged locomotion. The rest of this section provides information about our research on bipedal locomotion.

5.1 Motion Engine

Our motion engine has three different representation infrastructures, which allow different levels of abstractions.

The first infrastructure contains a data structure, named "Body", which stores the physical properties of the robot as well as the functions used for performing related calculations. The "Body" is composed of several "Link"s, which are data structures for storing the joint positions, angles, and related kinematic description parameters. Being platform independent and generic, this infrastructure makes it possible to define new robot platforms via some configuration files and allows controlling the joints of the robot platform easily.

The second infrastructure is a hierarchical one. A root engine is defined at the very top level and the common properties and functions of each different motion engine are included in it. Different motion engines are inherited from the root engine, and platform and motion algorithm specific parts are defined. For Nao, four main features are implemented. The first one is to make the robot perform a static walking in which the robot tries to keep the ground projection of its Center of Mass (CoM) inside the support polygon. In addition to the static walking feature, a dynamic walking feature is also implemented. A signal generation-based algorithm, which is very similar to Central Pattern Generator [12] is used. The motion of the head

is separated from the motion of the rest of the body and implemented as the third feature. The last feature is the motion player, which reads the sequential joint angles from pose definition files and plays them to realize some special actions, such as kicking the ball and standing up from a fallen position.

The third infrastructure is a container for some common motion-related functions. In addition to the matrix operations which are necessary for kinematics calculations, implementations to read configuration files are also included as common functions.

5.2 Implemented Biped Walking Algorithms

Walking algorithms can be classified into two main groups; static walking and dynamic walking. The main principle of static walking is to preserve stability all the time during the motion, which guarantees that the robot will not fall unless there is an external force to push the robot to fall. The most common method for static walking is based on keeping the CoM inside the support polygon. Although static walking is advantageous in terms of maintaining the balance continuously, the length of each step is very limited because of the concerns related to keeping the CoM inside the support polygon. Therefore, it is not possible to achieve a fast walk using static walking techniques. On the other hand dynamic walking is based on maintaining the balance by using dynamic properties of the motion. There are mainly three different methodologies;

- **Zero-Moment Point** (ZMP) Criterion [13] is very similar to CoM based static walking. The only difference is that holding ZMP inside the support polygon is enough to maintain the balance. Although it is a very common method, it is computationally very expensive.
- **Passive-Dynamic Walking** (PDW) [14] where CoM is carried along the motion direction, and the body moves along the motion direction because of the gravity. The important point is the timing of the foot contact of the swinging leg with the ground.
- Central Pattern Generators (CPGs) [12] method is based on the synchronous movement patterns of the joints. For this purpose, a signal is assigned for each joint, and the system is trained for synchronous patterns and balanced locomotion as a whole.

We implemented static walking with CoM criterion and a dynamic walking with CPG method [15] to be used on our Nao robots.

5.2.1 Static Walking with CoM Criterion

As a static walking example we implemented finite state machine based biped walking whose main criterion is to keep the CoM inside the support polygon. The implementation is essentially a combination of three different motions. The first motion is to shift the CoM into the sole of the support foot. This motion is generated on the sagittal plane and in the direction of the motion in order to prevent CoM from being left behind or beyond. By this way, it is provided that the ground projection of the CoM is kept inside the sole of the support foot. Foot pressure sensors (FPSs) are used for feedback. When it is guaranteed that the weight of the body is carried by the support foot, swinging leg is shortened. During this motion, the body is moved along the walking direction by the joints at the ankle of the support foot until the ground projection of the CoM is on the boundaries of the support foot. This makes it possible to use a longer step. As the last motion, swinging leg is moved in the direction of motion and landed on the ground. This point is where the robot is in its most unstable state. In order to maintain the stability of the robot, the landing speed of the foot and reactiveness to be aware of the landing moment of the swinging leg should be controlled. If the landing is too fast, the landing will not be noticed on time and the state transition would not be accomplished properly. This disturbs the balance through the support leg and the robot falls down on the support leg. On the other hand, if the landing is too slow, the motion will be very slow and inefficient. For these purposes, forward kinematics is used in addition to the FPSs. Using forward kinematics, relative locations of the feet are calculated and swinging foot is landed fast enough while it is horizontally far away from the support foot. When the vertical distance between swinging foot and the ground is small, landing speed is slowed down and FPSs are used to determine whether the swinging foot is landed or not. Because of the symmetric nature of the biped walking, after the swinging leg is landed, the roles are switched between legs and same motion is repeated for the other leg.

5.2.2 Dynamic Walking with CPG Method

As a dynamic walking example, a walking method based on that of the champion of the Humanoid League in the RoboCup07, NimbRo [15], is implemented. They defined three important features for each leg; leg extension, leg angle, and foot angle. Leg extension is the distance between hip joint and ankle joint. It determines the height of the robot while moving. Leg angle is the angle between the pelvis plate and the line from hip to ankle. It has three components; roll, pitch, and yaw. The third feature, foot angle, is defined as the angle between foot plate and pelvis plate. It has only two components; roll and pitch. Using these features helps us to have more

abstract calculations for the motion.

Before finding motion features, a central clock (ϕ_{trunk}) is generated for the trunk which is between $-\pi$ and π. Each leg is fed with a different clock (ϕ_{leg}) with $ls \times \pi/2$ phase shift where ls represents leg sign and it is -1 for the left leg while $+1$ for the right leg. The synchronization of the legs can be preserved in this way. In the calculations of motion features at a given time, the corresponding phase value is considered and the values for features are calculated by using these phase values.

In order to find the leg angle and foot angle features, motion at each step is divided into five sub-motions; *shifting*, *shortening*, *loading*, *swinging*, and *balance*. In shifting sub-motion, lateral shifting of the CoM is handled. For this purpose, a sinusoidal signal is used:

$$\theta_{shift} = ls \times a_{shift} \times sin(\phi_{leg}) \tag{5.1}$$

where a_{shift} is a constant to determine the shift amount. This shift signal is applied to the leg and the foot with different magnitudes. While it is applied to the leg as it is, only the half of the value with a negation is applied to the foot:

$$\theta_{LegShift} = \theta_{shift} \tag{5.2}$$

$$\theta_{FootShift} = -0.5 \times \theta_{shift} \tag{5.3}$$

. The second important sub-motion is shortening signal and it is not always applied. Phase value for shortening is calculated with

$$\phi_{short} = v_{short} \times (\phi_{leg} + \pi/2 + o_{short}) \tag{5.4}$$

. where v_{short} is a constant to determine the shortening duration, and o_{short} is also a constant for the phase shift of shortening relative to the shifting sub-motion. During shortening state, both a joint angle for the foot and a leg extension value are calculated by

$$\gamma_{short} = \{ \begin{array}{cc} -a_{short} \times 0.5 \times (cos(\phi_{short}) + 1) & if -\pi \leq \phi_{short} < \pi \\ 0 & otherwise \end{array} \tag{5.5}$$

and

$$\theta_{footShort} = \{ \begin{array}{cc} 0.5 \times (cos(\phi_{short}) + 1) & if -\pi \leq \phi_{short} < \pi \\ 0 & otherwise \end{array} \tag{5.6}$$

The third sub-motion of the step is loading which is also not always applied. The function of the corresponding phase value is given as

$$\phi_{load} = v_{load} \times piCut(\phi_{leg} + \pi/2 - \pi/v_{short} + o_{short}) - \pi \tag{5.7}$$

. where v_{load} is a constant to determine the load duration, and $piCut$ is a function which maps the value of the parameter to a value between $-\pi$ and π. In this phase, only a part of the leg extension is calculated as

$$\gamma_{load} = \{ \begin{array}{cc} -0.5 \times a_{load} \times cos(\phi_{load}) + 1) & if -\pi \leq \phi_{load} < \pi \\ 0 & otherwise \end{array} \tag{5.8}$$

where a_{load} is determined according to the motion speed. Swinging is the most important part of the motion. In this part, the leg is unloaded, shortened and moved along the way of motion which reduces the stability of the system considerably. This movement has effects on each component of the leg and the foot angle features of the motion. Because the behavior of the θ_{swing} is changed during the swinging process, another clock signal is calculated as

$$\phi_{swing} = v_{swing} \times \phi_{leg} + \pi/2 + o_{swing} \tag{5.9}$$

. where v_{swing} is a constant to determine the swinging duration, and o_{swing} is also a constant for a phase shift of swinging sub-motion. θ_{swing} is calculated by the equation:

$$\theta_{swing} = \{ \begin{array}{cc} sin(\phi_{swing}) & if -\pi/2 \leq \phi_{swing} < \pi/2 \\ b \times (\phi_{swing} - \pi/2 + 1) & \pi_{/2} \leq \phi_{swing} \\ b \times (\phi_{swing} + \pi/2 - 1) & otherwise \end{array} \tag{5.10}$$

where $b = -(2/(2 \times \pi \times v_{swing} - \pi)$.

According to the speed and direction of the motion, θ_{swing} is distributed to the components of leg and foot angle features as

$$\theta_{legSwing^r} = ls \times v_0 \times \theta swing \tag{5.11}$$

.

$$\theta_{legSwing^p} = v_1 \times \theta swing \tag{5.12}$$

.

$$\theta_{legSwing^y} = ls \times v_2 \times \theta swing \tag{5.13}$$

.

$$\theta_{footSwing^r} = ls \times 0.25 \times v_0 \times \theta swing \tag{5.14}$$

.

$$\theta_{footSwing^p} = 0.25 \times v_1 \times \theta swing \tag{5.15}$$

.

As the last component of the step, balance, correction values for the deviations of the other operations are added to the system from the foot angle feature and the rolling component of the leg angle feature. Correction equations are:

$$\theta_{footBalance^r} = ls \times 0.5 \times_0 \times cos(\phi leg + 0.35) \tag{5.16}$$

.

$$\theta_{footBalance^p} = 0.02 + 0.08 \times v_1 - 0.04 \times v_1 \times cos(2.0 \times \phi_{leg} + 0.7); \quad (5.17)$$

.

$$\theta_{legBalance^r} = 0.01 + ls \times v_0 + |v_0| + 0.1 \times v_2 \quad (5.18)$$

. At the end, the corresponding parts of the motion features can be combined. Equations to combine corresponding parts are given as

$$\theta_{leg^r} = \theta_{legSwing^r} + \theta_{legShift} + \theta_{legBalance^r} \quad (5.19)$$

.

$$\theta_{leg^p} = \theta_{legSwing^p} \quad (5.20)$$

.

$$\theta_{leg^y} = \theta_{legSwing^y} \quad (5.21)$$

.

$$\theta_{foot^r} = \theta_{footSwing^r} + \theta_{footShift} + \theta_{footBalance^r} \quad (5.22)$$

.

$$\theta_{foot^p} = \theta_{footSwing^p} + \theta_{footShort} + \theta_{footBalance^p} \quad (5.23)$$

.

$$\gamma = \gamma_{short} + \gamma_{load} \quad (5.24)$$

. There is a direct relation between the joints of the robot and these motion features. In other words, by using these features, the values for hip, knee, and ankle joints can be calculated easily using the following rules:

1. Since there is only one feature which changes the distance between the hip and ankle which is the leg extension, the knee joint is determined according to only the value of the leg extension feature of the motion. From five sub-motions of a step, the leg extension feature is calculated and its value is in between -1 and 0. While calculating the value of the knee joint from the extension, the normalized leg length, n, is calculated as $n = 1 + (1 - n_{min}) \times \gamma$, where $n_{min} = 0.875$. The knee joint value is directly proportional to the arc cosine of the length of the leg; $\theta_{knee} = -2 \times acos(n)$.

2. Another special case for the motion is its turn component. If we consider the physical construction of the robot, there is only one joint dedicated for this motion; hip yaw joint. So, the turn component of the motion directly determines the value for the hip yaw joint which means $\theta_{hip^y} = \theta_{leg^y}$.

3. There are two more joints at the hip. By definition, roll and pitch components of the leg angle feature directly determine the values of the hip joints. This means that $\theta_{hip^r} = \theta_{leg^r}$ and $\theta_{hip^p} = \theta_{leg^p}$. In order to compensate for the effects of the leg extension on the pitch component of the leg angle feature, half of the value of the knee joint angle should be rotated around the turn component of the motion and added to the hip pitch and roll joints. In other words, pitch and roll components of the hip joints can be calculated by

$$\begin{pmatrix} \theta_{hip^r} \\ \theta_{hip^p} \end{pmatrix} = \begin{pmatrix} \theta_{leg^r} \\ \theta_{leg^p} \end{pmatrix} + R_{\theta_{hip^y}} \times \begin{pmatrix} 0 \\ -0.5 \times \theta_{knee} \end{pmatrix} \tag{5.25}$$

4. The values of the ankle joints are calculated by applying the inverse of the rotation around the turn component of the motion to the difference between the leg and the foot angle features. In order to compensate for the effects of the leg extension on the pitch component of the foot angle feature, half of the value of the knee joint angle should also be added to the ankle pitch joint. In other words, pitch and roll components of the ankle joints can be calculated by

$$\begin{pmatrix} \theta_{ankle^r} \\ \theta_{ankle^p} \end{pmatrix} = \begin{pmatrix} 0 \\ -0.5 \times \theta_{knee} \end{pmatrix} + R^{-1}_{\theta_{hip^y}} \times \begin{pmatrix} \theta_{foot^r} - \theta_{leg^r} \\ \theta_{foot^p} - \theta_{leg^p} \end{pmatrix} \tag{5.26}$$

The resulting signals of each joint of a leg are given in figures 5.1, 5.2, 5.3, 5.4, 5.5, 5.6 for forward walking direction.

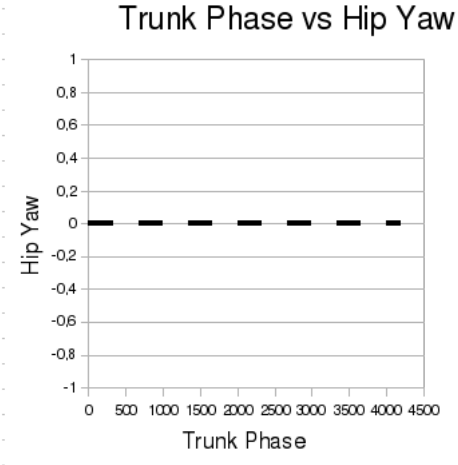


Figure 5.1: Signal for controlling the hip yaw joint for forward walking.

Aside from the implementation inspired from the work of the NimbRo team, we have also developed a custom algorithm for bipedal walking, which

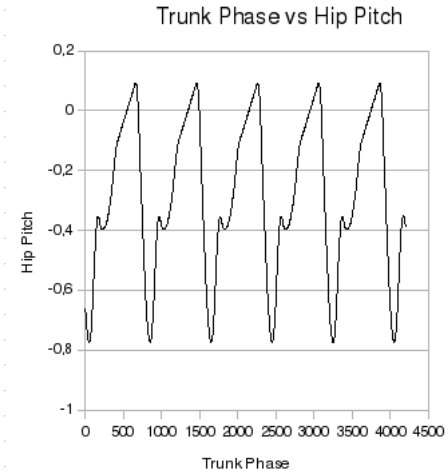


Figure 5.2: Signal for controlling the hip pitch joint for forward walking.

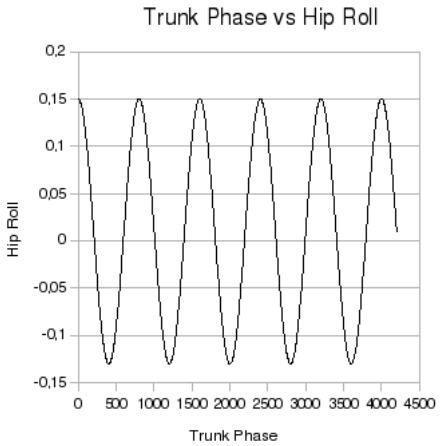


Figure 5.3: Signal for controlling the hip roll joint for forward walking.

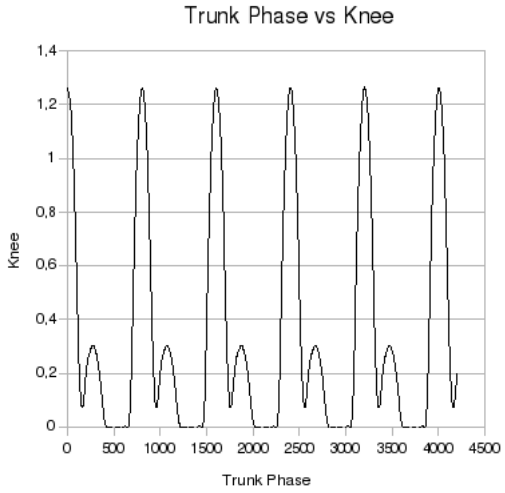


Figure 5.4: Signal for controlling the knee joint for forward walking.

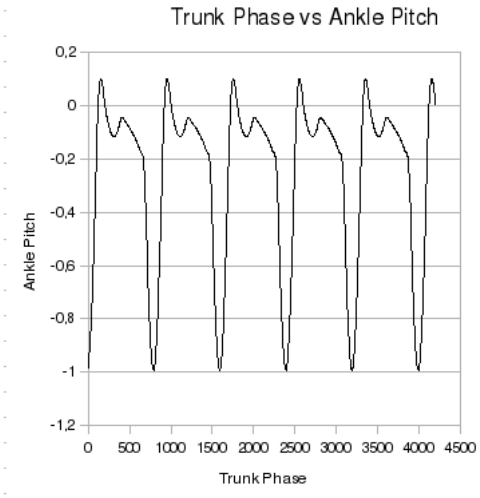


Figure 5.5: Signal for controlling the ankle pitch joint for forward walking.

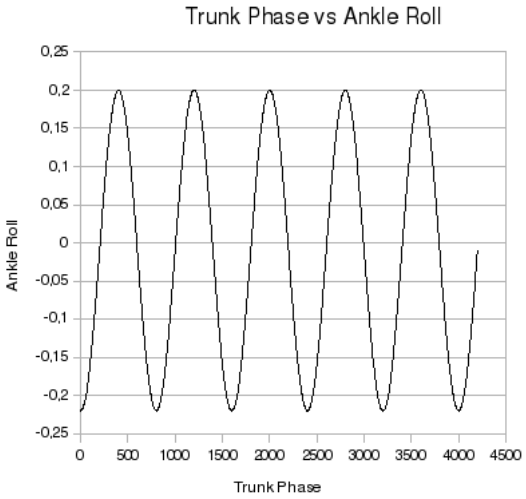


Figure 5.6: Signal for controlling the ankle roll joint for forward walking.

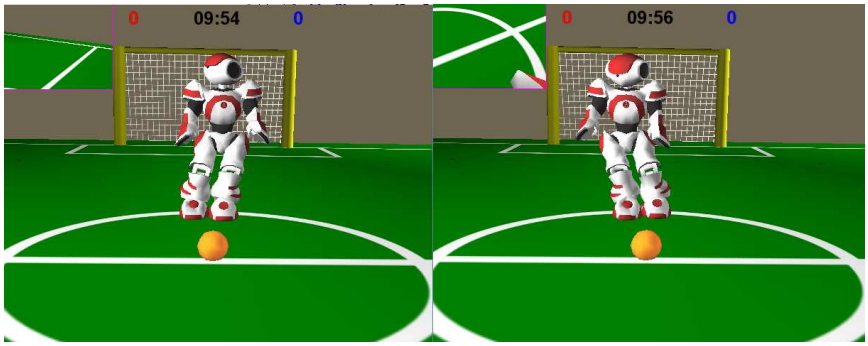


Figure 5.7: Robot swinging from one side to the other periodically by setting a sinusoidal angle value to the hip roll joint.

is also a CPG-based method. In our design, the main walking motion starts from the hip, specifically the roll joint, which makes the body to swing from one side to the other. In order to keep the feet parallel to the ground while swinging, the ankle roll joint angles should be set to the negative of the value of the corresponding hip roll joint angle. The periodic movement of the hip is obtained by using a sinusoidal signal to be supplied as the hip roll joint angle. Figure 5.7 illustrates the robot swinging from one side to the other periodically.

In order to realize this movement, the hip roll and ankle roll angles are set according to the following equations.

$$\theta_{hip_{roll}} = A_{hip_{roll}} sin(period)$$
$$\theta_{ankle_{roll}} = -A_{ankle_{roll}} sin(period)$$

This motion is the basis of the entire walking since it passes the projection of the center of mass from one foot to the other periodically, letting the idle foot to move according to the requested motion command.

In order to make the robot perform a stepping motion, the pitch angles on the leg chain should be set. These angles again take sinusoidal values which are consistent with the hip roll angle. The following equations illustrate how the values of these angles are computed.

$$\begin{aligned}\theta_{hip_{pitch}} &= A_{pitch} sin(period) + \theta_{hip^{rest}_{pitch}} \\ \theta_{knee_{pitch}} &= -2A_{pitch} sin(period) + \theta_{knee^{rest}_{pitch}} \\ \theta_{ankle_{pitch}} &= A_{pitch} sin(period) + \theta_{ankle^{rest}_{pitch}}\end{aligned}$$

The A_{pitch} value determines how big the step is going to be. Figure 5.8 shows the robot walking forwards. Obtaining backwards walk does not require much work but just reversing the iteration of the *period* value, which is defined as $0 < period < 2\pi$.

Similarly, making the robot move laterally is possible by setting the *roll* angles instead of the *pitch* angles together with the knee pitch, while turning around is possible by setting the $hipYawPitch$ joint angles properly. The amplitudes $A_{pitch}, A_{roll}, A_{yaw}$ are multiplied with the corresponding motion component, namely $forward, left, turn$ which are normalized in the interval $[-1, 1]$, to manipulate the velocity of the motion. In order to make the robot move omnidirectionally, the sinusoidal signals that are computed individually for each motion component are summed up and the final joint angle values obtained in that way. For instance, it is possible to make the robot walk diagonally in the north-west direction by simply assigning positive values to both the $forward$ and the $left$ components.

Figure 5.8: Robot walking forward by moving its leg joints in harmony with each other.

Chapter 6

Planning and Behaviors

The soccer domain is a continuous environment, but the robots operate in discrete time steps. At each time step, the environment, and the robots' own states change. The planner keeps track of those changes, and makes decisions about the new actions. Therefore, first of all, the main aim of the planner should be sufficiently modeling the environment and updating its status. Second, the planner should provide control inputs according to this model.

We have developed a four layer planner model, that operates in discrete time steps, but exhibits continuous behaviors, as shown in Figure 6.1

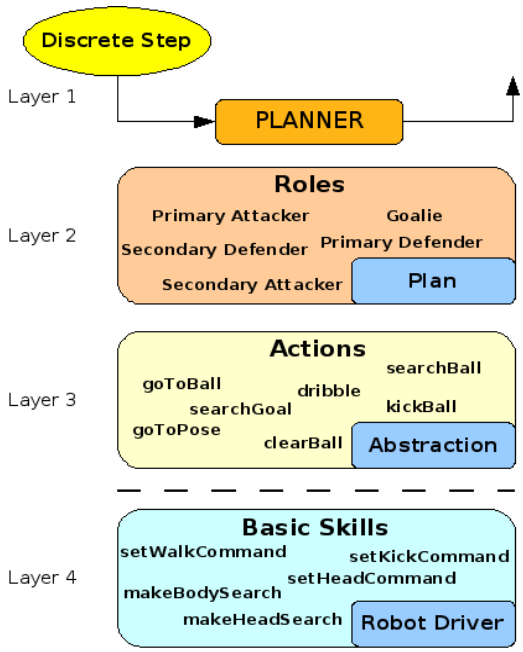


Figure 6.1: Multi-layer Planner.

The topmost layer provides a unified interface to the planner object. The second layer deals with different roles that a robot can take. Each role incorporates an "Actor" using the behaviors called "Actions" that the third layer provides. Finally, the fourth layer contains basic skills that the actions of the third layer are built upon. A set of well-known software design concepts like *Factory Design Pattern*[16], *Chain of Responsibility Design Pattern* [17] and *Aspect Oriented Programming* [18].

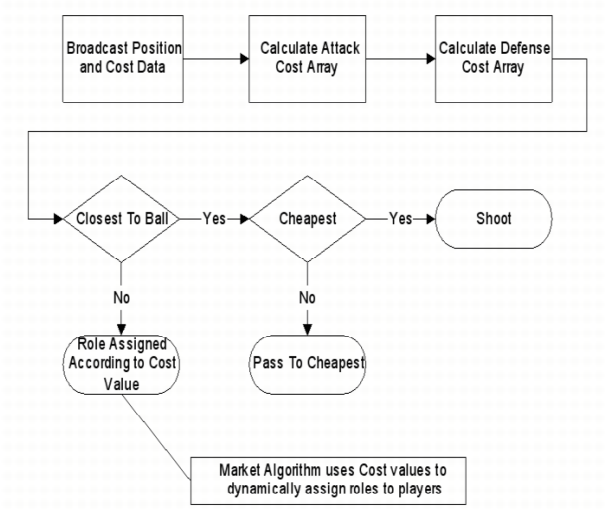


Figure 6.2: Flowchart for task assignment.

For coordination among the teammates and task allocation, we employ a market driven task allocation scheme [19, 20]. In this method, the robots calculate a cost value (their fitness) for each role. The calculated costs are broadcasted through the team and based on a ranking scheme, the robots chose the most appropriate role for their costs. Here, each team member calculates costs for its assigned tasks, including the cost of moving, aligning itself suitably for the task, and the cost of object avoidance, then looks for another team member who can do this task for less cost by opening an auction on that task. If one or more of the robots can do this task with a lower cost, they are assigned to that task, so both the robots and the team increase their profit. Other robots take actions according to their cost functions (each takes the action that is most profitable for itself). Since all robots share their costs, they know which task is appropriate for each one so they do not need to tell others about their decisions and they do not need a leader to assign tasks. If one fails, another would take the task and go on working.

The approach is shown in the flowchart given in Figure 6.2. The robot with the smallest score cost C_{ES} will be the primary attacker. Similarly

the robot, except the primary attacker, with the smallest $C_{defender}$ cost will be the defender. If $C_{auctioneer}$ is higher than all passing costs ($C_{bidder(i)}$) then the attacker will shoot, else, it will pass the ball to the robot with the lowest $C_{bidder(i)}$ value. The cost functions used in the implementations are as follows:

$$C_{ES} = \mu_1.t_{dist} + \mu_2.t_{align} + \mu_3.clear_{goal} \quad (6.1)$$

$$C_{bidder(i)} = \mu_1.t_{dist} + \mu_2.t_{align} + \mu_3.clear_{teammate(i)} + C_{ES(i)}, i \neq robotid \quad (6.2)$$

$$C_{auctioneer} = C_{ES(robotid)} \quad (6.3)$$

$$C_{defender} = \mu_5.t_{dist} + \mu_6.t_{align} + \mu_7.clear_{defense} \quad (6.4)$$

where $robotid$ is the id of the robot, t_{dist} is the time required to move for specified distance, t_{align} is the time required to align for specified amount, μ_i are the weights of several parameters to emphasize their relative importance in the total cost function, $clear_{goal}$ is the clearance from the robot to goal area-for object avoidance, $clear_{defense}$ is the clearance from the robot to the middle point on the line between the middle point of own goal and the ball-for object avoidance, and similarly $clear_{teammate(i)}$ is the clearance from the robot to the position of a teammate. Each robot should know its teammates score and defense costs. In our study each agent broadcasts its score and defense costs. Since the auctioneer knows the positions of its teammates, it can calculate the $C_{bidder(id=robotid)}$ value for its teammates.

The game strategy can easily be changed by changing the cost functions in order to define the relative importance of defensive behavior over offensive behavior, and this yields greater flexibility in planning, which is not generally possible.

Acknowledgements

This work is supported by Boğaziçi University Research Fund through project 06HA102 and TUBITAK Project 106E172.

Bibliography

[1] Robocup. www.robocup.org/.

[2] SPL. Robocup standard platform league www.tzi.de/spl/.

[3] Aldebaran-Nao. http://www.aldebaran-robotics.com/eng/nao.php.

[4] Trolltech's Qt Development Framework. http://trolltech.com/products.

[5] Ethem Alpaydın. *Machine Learning*. MIT Press, 2004.

[6] Thomas Lemaire, Cyrille Berger, Il-Kyun Jung, and Simon Lacroix. Vision-based slam: Stereo and monocular approaches. *Int. J. Comput. Vision*, 74(3):343–364, 2007.

[7] H. Köse, B. Çelik, and H. L. Akın. Comparison of localization methods for a robot soccer team. *International Journal of Advanced Robotic Systems*, 3(4):295–302, 2006.

[8] K. Kaplan, B. Çelik, T. Meriçli, Ç. Mericli, and H. L. Akın. Practical extensions to vision-based monte carlo localization methods for robot soccer domain. *RoboCup 2005: Robot Soccer World Cup IX, LNCS*, 4020:420–427, 2006.

[9] H. Köse and H. L. Akın. The reverse monte carlo localization algorithm. *Robotics and Autonomous Systems*, 55(6):480–489, 2007.

[10] Greg Welch and Gary Bishop. An introduction to the kalman filter. Technical report, Chapel Hill, NC, USA, 1995.

[11] Sebastian Thrun, Dieter Fox, Wolfram Burgard, and Frank Dellaert. Robust monte carlo localization for mobile robots. *Artificial Intelligence*, 128(1-2):99–141, 2001.

[12] C. Pinto and M. Golubitsky. Central pattern generators for bipedal locomotion. *J Math Biol*, 2006.

[13] M. Vukobratovic and D. Juricic. Contribution to the synthesis of biped gait. *IEEE Transactions on Bio-Medical Engineering*, 1969.

[14] Tad McGeer. Passive dynamic walking. *The International Journal of Robotics Research*, 9:62–82, April 1990.

[15] Sven Behnke. Online trajectory generation for omnidirectional biped walking. 2006.

[16] Wikipedia Anonymous. Factory method pattern, 2007.

[17] Wikipedia Anonymous. Chain-of-responsibility pattern, 2007.

[18] Wikipedia Anonymous. Aspect-oriented programming, 2007.

[19] H. Köse, Ç. Meriçli, K. Kaplan, and H. L. Akın. All bids for one and one does for all: Market-driven multi-agent collaboration in robot soccer domain. *Computer and Information Sciences-ISCIS 2003, 18th International Symposium Antalya, Turkey, Proceedings LNCS 2869*, pages 529–536, 2003.

[20] H. Köse, K. Kaplan, Ç. Meriçli, U. Tatlıdede, and H. L. Akın. Market-driven multi-agent collaboration in robot soccer domain. *Cutting Edge Robotics*, pages 407–416, 2005.