Technical University of Crete, Greece
Department of Electronic and Computer Engineering

Field Landmark Recognition and Localization for the Robotstadium Online Soccer Competition

George Georgakis

Thesis Committee
Assistant Professor Michail G. Lagoudakis (ECE)
Assistant Professor Georgios Chalkiadakis (ECE)
Professor Michalis Zervakis (ECE)

Chania, October 2012

Πολυτεχνειο Κρητης

Τμημα Ηλεκτρονικων Μηχανικων και Μηχανικων Υπολογιστων

Αναγνώριση Χαρακτηριστικών Γηπέδου και Εντοπισμός Θέσης για τον Διαδικτυακό Διαγωνισμό Ποδοσφαίρου **RobotStadium**

Γιώργος Γεωργάκης

Εξεταστική Επιτροπή

Επίκουρος Καθηγητής Μιχαήλ Γ. Λαγουδάκης (ΗΜΜΥ)

Επίκουρος Καθηγητής Γεώργιος Χαλκιαδάκης (ΗΜΜΥ)

Καθηγητής Μιχάλης Ζερβάκης (ΗΜΜΥ)

Χανιά, Οκτώβριος 2012

Abstract

Visual object recognition is a key ability for autonomous robotic agents in dynamic and partially observable environments, such as the RoboCup competition, where teams of autonomous robots compete against each other in soccer games. Reliable object recognition is also crucial in achieving reliable self-localization, namely awareness of the self-location in the field at all times using the fixed field landmarks. This thesis focuses on the Robot-Stadium Online Soccer Competition which is a simulation of the RoboCup Standard Platform League (SPL) based on the Webots robot simulator. Our work aims at developing a dependable visual landmark recognition module and on top of that a dependable self-localization module. Using a variety of image processing techniques, our landmark recognition method successfully detects several field landmarks from the camera images: the blue and the yellow goals, their left and right goal posts, the center circle, the four field corners, the four goal area corners, and the two line junctions along the middle field line. From any position in the field, using a wide horizontal scan, our robot player is able to accurately detect several landmarks, disambiguate them, and estimate the distance and direction to each one of them. Subsequently, a constraint-based geometric localization method pinpoints the agent's location (position and orientation) in the field by exploiting the constraints imposed by the observed landmarks. Our experimental results demonstrate that our approach leads to accurate self-localization with the average error from the true location in position and orientation ranging from $12cm/4^\circ$ in the empty $4m \times 6m$ SPL field to about $42cm/23^\circ$ in the worst case of a full game, where visual obstructions and misplacement due to pushing by other players are highly likely. These promising results and the simplicity of our approach make it potentially suitable for on-board execution on the real Aldebaran Nao humanoid robots used in the RoboCup Standard Platform League (SPL).

Περίληψη

Η οπτική αναγνώριση αντικειμένων αποτελεί βασική ικανότητα για αυτόνομους ρομποτικούς πράκτορες μέσα σε δυναμικά και μερικώς παρατηρήσιμα περιβάλλοντα, όπως ο διαγωνισμός RoboCup, όπου ομάδες αυτόνομων ρομπότ ανταγωνίζονται μεταξύ τους σε αγώνες ποδοσφαίρου. Η αξιόπιστη αναγνώριση αντικειμένων είναι επίσης ζωτικής σημασίας για την επίτευξη αξιόπιστου εντοπισμού θέσης, δηλαδή για τη συνειδητοποίηση της θέσης του πράκτορα μέσα στο γήπεδο ανά πάσα στιγμή χρησιμοποιώντας τα σταθερά χαρακτηριστικά του γηπέδου. Η παρούσα εργασία επικεντρώνεται στον διαδικτυακό διαγωνισμό ποδοσφαίρου RobotStadium ο οποίος είναι μια προσομοίωση του RoboCup Standard Platform League (SPL) βασισμένη στον προσομοιωτή ρομποτικών συστημάτων Webots. Η εργασία μας στοχεύει στην ανάπτυξη μιας αξιόπιστης μεθόδου οπτικής αναγνώρισης χαρακτηριστικών γηπέδου και βάσει αυτής μιας αξιόπιστης μεθόδου εντοπισμού θέσης. Χρησιμοποιώντας διάφορες τεχνικές επεξεργασίας εικόνας, η προτεινόμενη μέθοδος αναγνώρισης χαρακτηριστικών ανιχνεύει επιτυχώς διάφορα χαρακτηριστικά του γηπέδου μέσα από τις εικόνες της κάμερας: το μπλε και το κίτρινο τέρμα, τα αριστερά και δεξιά δοκάρια τους, τον κεντρικό κύκλο, τις τέσσερις γωνίες του γηπέδου, τις τέσσερις γωνίες στις περιοχές τέρματος και τους δύο κόμβους γραμμών κατά μήκος της μεσαίας γραμμής του γηπέδου. Από οποιαδήποτε θέση στο γήπεδο, χρησιμοποιώντας μια ευρεία οριζόντια σάρωση, ο ρομποτικός παίκτης μας είναι σε θέση να ανιχνεύσει με ακρίβεια πολλά χαρακτηριστικά του γηπέδου, να τα αποσαφηνίσει και να εκτιμήσει την απόσταση και την διεύθυνση για κάθε ένα από αυτά. Στη συνέχεια, μία μέθοδος εντοπισμού θέσης βασισμένη σε γεωμετρικούς περιορισμούς προσδιορίζει τη θέση του παίκτη (συντεταγμένες και προσανατολισμός) στο γήπεδο αξιοποιώντας τους περιορισμούς που επιβάλλονται από τα παρατηρούμενα χαρακτηριστικά γηπέδου. Τα πειραματικά μας αποτελέσματα δείχνουν ότι η προσέγγισή μας οδηγεί σε ακριβή εντοπισμό θέσης με το μέσο σφάλμα από την πραγματική θέση (συντεταγμένες και προσανατολισμός) να κυμαίνεται από $12cm/4^\circ$ στο άδειο $4m \times 6m$ γήπεδο SPL σε περίπου $42cm/23^\circ$ στη χειρότερη περίπτωση ενός πλήρους αγώνα, όπου οπτικά εμπόδια και ανεπιθύμητες μετατοπίσεις από άλλους παίκτες είναι πολύ πιθανά. Αυτά τα ενθαρρυντικά αποτελέσματα και η απλότητα της προσέγγισής μας την καθιστούν δυνητικά κατάλληλη για υλοποίηση και εκτέλεση πάνω στο ανθρωποειδές ρομπότ Aldebaran Nao που χρησιμοποιείται στο RoboCup Standard Platform League (SPL).

Acknowledgements

First of all I would like to thank my advisor Michail G. Lagoudakis for his mentoring during this thesis. I really appreciate his dedication and trust demonstrated towards myself.

To all my friends here in Chania, I am happy we had the chance to spend these past few years together. Special thanks to my colleague and friend Konstantinos.

Finally, I am deeply grateful towards my family for their support throughout the duration of my education studies in Chania. I know that despite the difficulties, for them I was always the priority.

Ευχαριστίες

Καταρχήν θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου Μιχαήλ Γ. Λαγουδάκη για την καθοδήγησή του κατά την διάρκεια της εργασίας. Εκτιμώ πραγματικά την αφοσίωσή του και την εμπιστοσύνη που επέδειξε προς τον εαυτό μου.

Για όλους τους φίλους μου εδώ στα Χανιά, είμαι χαρούμενος που είχαμε την ευκαιρία να περάσουμε τα τελευταία χρόνια μαζί. Ιδιαίτερες ευχαριστίες στον συμφοιτητή μου και φίλο μου Κωνσταντίνο.

Τέλος, είμαι βαθύτατα ευγνώμων προς την οικογένεια μου για την υποστήριξη τους καθ΄ όλη την διάρκεια της φοίτησής μου στα Χανιά. Ξέρω ότι παρά τις δυσκολίες, γι΄ αυτούς ήμουν πάντα η προτεραιότητα.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Robotic football competitions in the last decade are met with great enthusiasm and success. Being a very popular sport as it is, football turned out to be the best incentive for researchers and teams from all over the world who are attracted to participate to the annual competition of RoboCup Soccer. Through its various leagues, RoboCup Soccer intends to promote Robotics and Artificial Intelligence research in addition to the study of numerous subjects, such as multi-agent systems in dynamic environments and computer vision. Every team consists of fully autonomous robotic agents. This means that the football players have to decide and act on their own during the game, without any external control.

Robotstadium is a simulation of one of the most favoured leagues, the Standard Platform League (SPL), where all teams compete with the same robotic platform, the Aldebaran Nao humanoid robot. Therefore, software development is the main focus for each team in order to build up its autonomous agents. This process can be complicated on account of the basic Artificial Intelligence problems that need to be solved and applied to the agent, such as perception of the environment, decision making under uncertainty, agent self-localization, multi-agent planning, and team coordination.

Dependable vision and self-localization modules provide crucial abilities to a robotic football player. The former is imperative for perceiving the environment and recognizing objects, while the latter provides an estimated position for the agent in the field, including its orientation. Both offer the opportunity for enhancing the abilities of the agents and the team, as they make possible the development of team coordination and strategy.

1.1 Thesis Contribution

This thesis presents a vision module for landmark recognition in the Robotstadium field and a self-localization module that exploits the recognized landmarks. Our vision module is able to recognize and locate field landmarks efficiently, in the interest of providing dependable information to the self-localization process. Initially, we process all pixels in the image, to achieve colour segmentation. This process is imperative as it reduces the information we obtain from the image into a specific number of colours. Based on that information, we detect objects of interest on the field. If a concentration of blue or yellow pixels is found, then a goal is detected within our visible area of the field. Furthermore, we recognize field lines by running vertical scan lines on the segmented image to locate possible white areas that correspond to field lines segments. Then, a filtering procedure is responsible to determine which of these segments are actually parts of a field line, discarding all segments corresponding to other objects, such as robot bodies. Subsequently, our method forms a set of detected lines by grouping the remaining segments and identifies the line type using a least-squares fitting method. The data sent to the self-localization module for each recognized landmark are the landmark type, its distance and its direction from robot and the landmark's fixed coordinates on the field. Our approach to the self-localization problem avoids time-consuming probabilistic methods and focuses on a constraint-based approach. Based on the detected landmarks, we are estimating our position by calculating the intersection points of virtual circles on the field, corresponding to candidate self-positions. After we collect a number of such intersection points, an algorithm determines which one is the best candidate, depending on the direction angles of the detected landmarks. Our numerous simulations provide us with encouraging results about our method's efficiency. Even in full game scenarios, where our agent has low visibility of landmarks, our estimates of self-position are satisfyingly close to the true position of the player.

1.2 Thesis Overview

Chapter 2 presents all the background information needed for this thesis, such as a description of the RoboCup Soccer competition and the RobotStadium online competition, as well as the necessary mathematical background. In Chapter 3 we state our problem and

we discuss the difficulties we had to overcome in our approach. In Chapter 4 we describe the field landmark recognition procedure for each individual landmark, while in chapter 5 we explain thoroughly our self-localization method. In Chapter 6 we demonstrate several scenarios to prove our self-localization method's effectiveness. In Chapter 7 we refer to related work by other teams on landmark recognition and localization in simulated soccer. Finally, Chapter 8 acts as an epilogue for this thesis, presenting our conclusions along with future improvements of our work.

Chapter 2

Background

2.1 RoboCup Soccer

RoboCup Soccer is the main division of the international robotics competition RoboCup [1]. It was founded in 1997 and aims at promoting the research fields of artificial intelligence, multi-agent systems and robotics, focusing in the game of football. All agents must be fully autonomous and able to function into dynamic environments, solving the issues of robotic cooperation, cognition and behavior. The ultimate goal as stated by the RoboCup Federation is:

> "By mid-21st century, a team of fully autonomous humanoid robot soccer players shall win the soccer game, comply with the official rule of FIFA, against the winner of the most recent World Cup" [2].

RoboCup Soccer includes the leagues of Humanoid, Middle-Size, Small-Size, Standard Platform and Simulation which are going to be presented shortly. Besides RoboCup Soccer, the RoboCup competition embodies the leagues of RoboCup Rescue, RoboCup @Home and RoboCup Junior with great success and world-wide participation.

2.1.1 Humanoid League

In this league, robots have human-like bodies and human-like senses. Challenges are the development of human-like motions, visual perception, self-localization and coordination. The league is divided into three subleagues depending on robot sizes; Kid, Teen and Adult. Robot models of these three subleagues are featured in Figure 2.1.

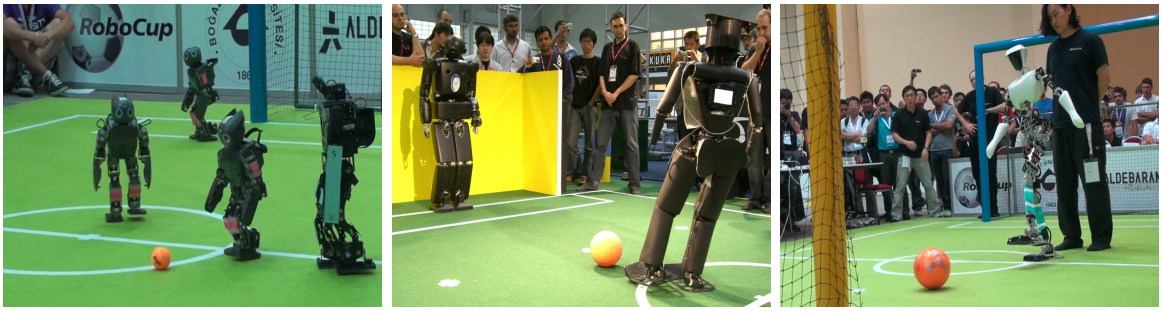

Figure 2.1: Humanoid League: Kid (left), Teen (center), and Adult (right) players.

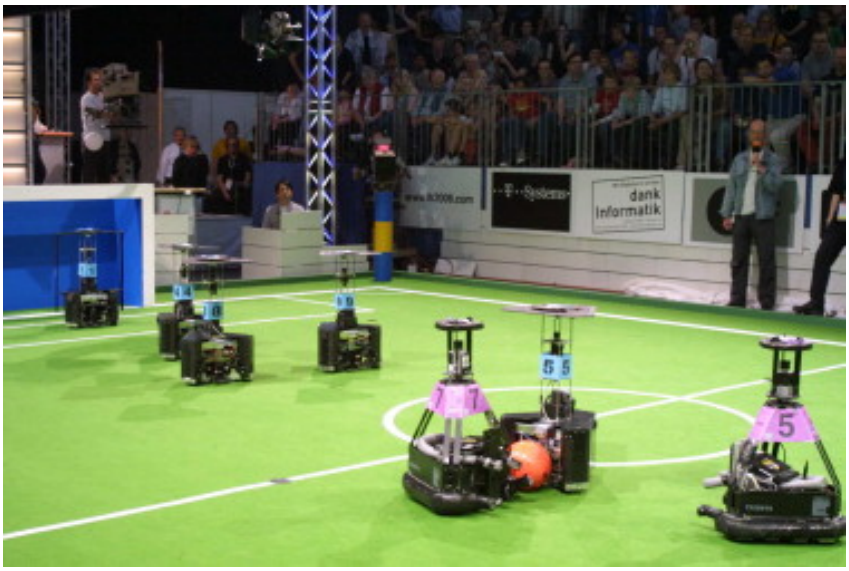

Figure 2.2: Instance of a Middle-Size League game.

2.1.2 Middle-Size League

Teams of six, up to 50cm diameter, robots are competing in this league with a FIFA regular size ball in a scaled human soccer field. Figure 2.2 presents an example.

2.1.3 Small-Size League

Robot players must fit within a 18cm cylinder that is 15cm tall in order to qualify for this league. Every team has five members and the game is played on a 6.05m long by 4.05m wide field with an orange golf ball. Figure 2.3 shows a highlight of a Small Size League game.

Figure 2.3: Instance of a Small-Size League game.

2.1.4 Standard Platform League

Standard Platform League teams use the same robot. Aldebaran's Nao humanoid robot replaced the former standard robot of Sony's AIBO since 2008. Teams concentrate on software development while trying to solve several problems such as dynamic motion, computer vision, perception, self-localization and ball-localization. Greece is being represented by team "Kouretes" [3], based at the Technical University of Crete. Figure 2.4 presents a game of Standard Platform League.

2.1.5 Simulation League

One of the oldest leagues in RoboCup Soccer focusing on team coordination and strategy. The game is played inside a computer by software agents on a virtual field. There are two subleagues; 2D and 3D which are presented in Figure 2.5.

Figure 2.4: Standard Platform League game.

Figure 2.5: Simulation examples of 2D League (left) and 3D League (right).

2.2 RobotStadium Competition

The RobotStadium Competition [4] is an online contest simulating the RoboCup Standard Platform League (SPL). It runs unofficially during the SPL, following its rules and regulations. The idea was for everyone to be able to program a team of robots (Selection between Nao and Darwin) to play football. RobotStadium's main focus is the use of

Figure 2.6: The RobotStadium competition environment.

robot's vision and sensors to create a player agent, unlike RoboCup3D Simulation League which is more focused on team coordination and strategy in the game. The competition offers many challenges because there are numerous areas that need development, just like the RoboCup SPL competition. A contestant has to implement computer vision, humanoid locomotion, a perception module using the robot's sensors, and a decision making algorithm to cover just the basics for his team [5] [6]. Figure 2.6 is an example of the RobotStadium environment.

2.2.1 Webots

WeBots is a professional development environment for robotic simulation. It started in 1996 at the Swiss Federal Institute of Technology (EPFL) by the Cyberbotics company [7]. Webots allows the researcher to build worlds with real physical properties in order to place a robot into certain conditions and includes a large collection of freely modifiable robot models, plus the ability to create one of your own. It also provides a

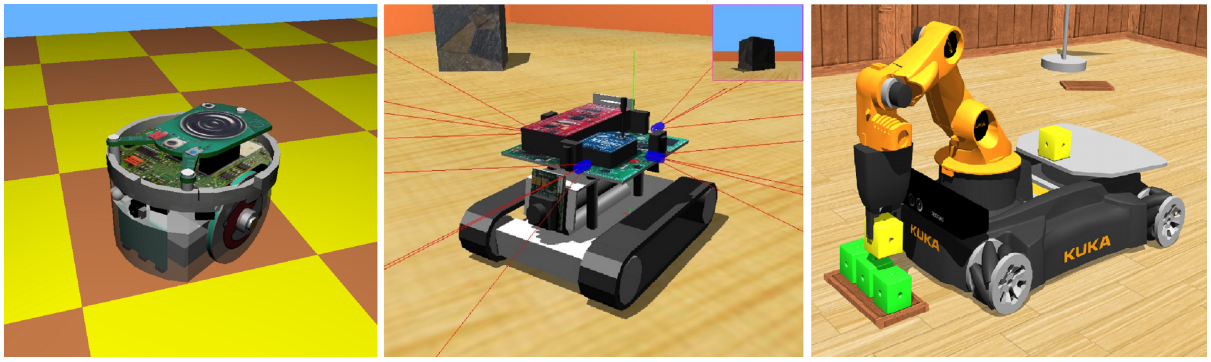

Figure 2.7: Examples of WeBots robotic models.

variety of sensors and actuators which are usually used in robotic experiments (e.g. accelerometers, proximity sensors, GPS, compass and gyro). The sensors are programmed to resemble as much as possible those of real robots, including the deviation of measurements. Webots is the platform in which the RobotStadium competition was built upon and it is still the developing tool for anyone who wants to enter the contest. Figure 2.7 shows examples of robotic modeling in WeBots.

2.2.2 Node Architecture

Here we will examine the nodes and fields of the Webots world that is used for RobotStadium. Figure 2.8 offers a chart of these nodes. In this chart, an arrow between two nodes represents an inheritance relationship. The inheritance relationship indicates that a derived node (at the arrow tail) inherits all the fields and API functions of a base node (at the arrow head). As mentioned above, Webots provides functionality for some sensors, that derive from the class Solid. The class Device which is in between, is actually an abstract class used to group common fields and functions such as the sensors we need for RobotStadium. A Solid node represents an object with physical properties such as dimensions, a contact material and optionally a mass. Therefore, a Robot node, which is used as basis for building any kind of robot, also derives from Solid, and in turn robot is extended by Supervisor and Player. Supervisor is a special kind of robot which is specially designed to control the simulation. In the RobotStadium case, supervisor controls the game sequence. Player is the robot team member and Player Camera is a class that inherits all fields and functions from Camera device, in order to be used for an

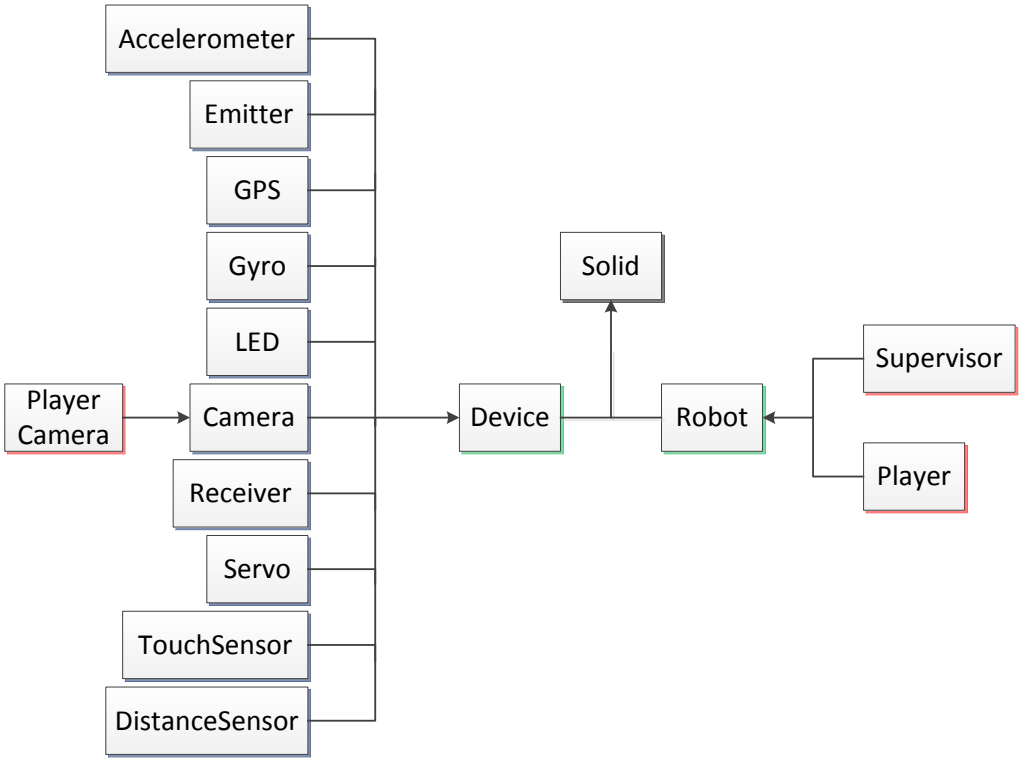

Figure 2.8: Webots node chart for the RobotStadium competition.

agent's vision. Finally, the program that is developed to describe a robot in a simulation, is called the controller.

2.2.3 Sensors and Actuators

Sensor nodes used in RobotStadium have been mentioned in the previous subsection, but have not yet been described in details. Only the camera was utilized for our work on landmark recognition and self-localization; the rest of the sensors were needed in the creation of a simple agent that plays soccer.

- **Camera** Explicit analysis of the camera node is carried out in the next chapter.
- **Accelerometer** Measures acceleration and gravity-induced reaction forces over x, y and z axes. It can be used for example to detect a fall or the up/down direction.

- **Emitter** Sends data, but cannot receive. Useful for team coordination in the game.
- **Receiver** Receives data, but cannot send. It is used with the Emitter node to simulate unidirectional or bidirectional communication between two robots.
- **GPS** Models a Global Positioning System (GPS) sensor which can obtain information about its absolute position in the environment.
- **Gyro** Measures, in radians per second $[rad/s]$, the angular velocity over x, y and z axes.
- **LED** Models a light emitting diode. The light produced by a LED can be used for debugging or informational purposes. Nao robot has LEDs on various body locations.
- **Servo** Is used to add a joint (1 degree of freedom (DOF)) in a mechanical simulation. The joint can be active or passive. The servo can be of linear (prismatic) or rotational (revolute) type. All joints on the Robot are accessed from the Servo node.
- **TouchSensor** Models a bumper or a force sensor. The TouchSensor comes in three different types. The "bumper" type simply detects collisions and returns a boolean status, the "force" type measures the force exerted on the sensor's body on one axis (z-axis) and finally the "force-3d" type measures the 3d force vector exerted by external object on the sensor's body.
- **Distance Sensor** Is used to model an infra-red sensor, a sonar sensor, or a laser range-finder. This device simulation is performed by detecting the collisions between one or several sensor rays and the bounding objects of Solid nodes in the environment.

All sensors and actuators produce values during the simulation. For those values to be accessed a `step` function call is employed which synchronizes the controller's data with the simulator and must be called at regular intervals. It retrieves all sensor data at the specified time of the call.

2.2.4 Robot Model

RobotStadium offers the option of choosing between two robot models for your team; the most commonly used Nao and recently added Darwin-OP. For the purposes of this thesis Nao was used. The real Nao robot was developed by Aldebaran Robotics [8] as an autonomous programmable humanoid robot mainly purposed for research and software development. It weighs 4.3Kg and has a height of 58cm. Nao V4 (Last Generation) has 60 to 90 minutes autonomy, 25 degrees of freedom, two HD cameras (1288×968 pixels), an Intel Atom CPU, and a variety of sensors.

Nao V3R (RoboCup Edition) is the Nao edition chosen as the platform for the Standard Platform League since RoboCup 2010. Although RobotStadium uses the NaoV3R model as well, there are three key differences compared to the real one. First, the real NaoV3R has two cameras situated on its head with a resolution of 640×480 pixels; one above the eyes and the other below. This was done so that the cameras have two different field of views for better vision, but they cannot operate simultaneously, only one at a time. The simulated robot has one camera of 160×120 resolution, but with a selection of high/low positions, which have an offset angle of 40 degrees, simulating the two cameras of the real Nao. Secondly, the real NaoV3R robot has 21 degrees of freedom, whereas the simulated one has 22. The extra degree is located at the pelvis giving the ability for the two hipYawPitch joints (left and right) to move independently, whereas these two joints on the real Nao are coupled and always take the same value. Furthermore, there are two extra ultrasound sensors on the simulated Nao, situated at the lower body of the robot, in addition to the two already existing on the upper body. The ultrasound sensors do not detect parts of the host robot unlike the real NaoV3R where in multiple occasions a robot's hand can be detected.

The rest of the sensors and capabilities are the same on both robots. A 3-axes Accelerometer, a 2-axes Gyro, 4 Foot Bumpers (2 on each foot), 8 Force Sensitive Resistor (FSR) sensors (4 on each foot), LEDs on eyes, chest, ears and feet and both have no actuated hands. Figure 2.9 displays the two robot models side by side.

2.2.5 Field and Supervisor

As said above, the RobotStadium Competition follows the rules and regulations of SPL [9], therefore the field dimensions and landmarks are always imitating those of

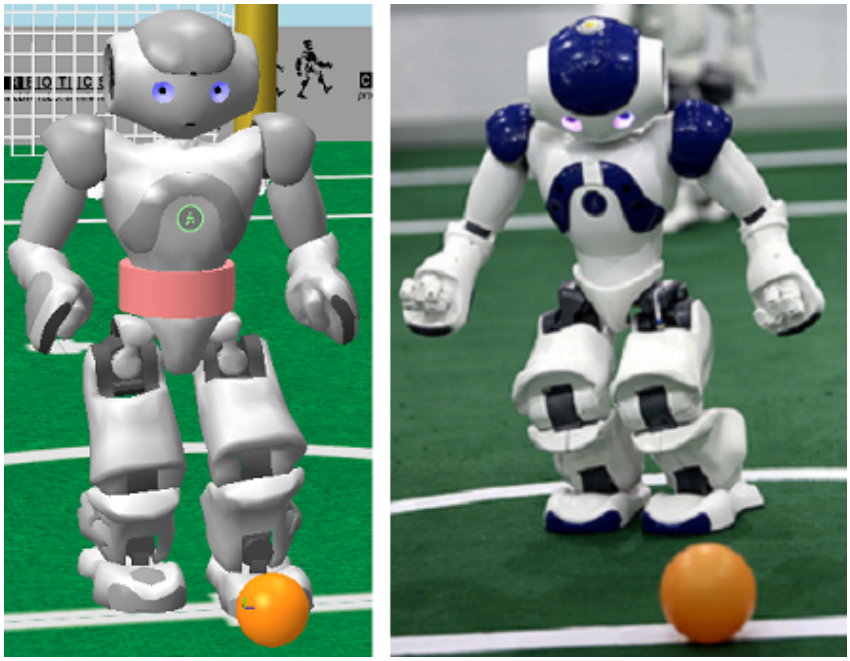

Figure 2.9: RobotStadium simulated Nao (left) and real Nao (right) robots.

SPL. But, because the 2012 Robotstadium contest did not take place, the rules of 2011 remained and were followed throughout this thesis. The most significant difference is the goal colors; in 2011 there were a yellow and a blue goal, unlike in 2012, where both goals are yellow. The field is 6m long, 4m wide and the center circle has a diameter of 1.2m. Figure 2.10 is a detailed representation of the specifications of the playing field.

The game is controlled by a program called supervisor. It is responsible for time keeping, state sequence, halftime changes, fouls, kick-off and robot and ball positions. The game is played in two periods of 10 minutes each. When the first period is over, the supervisor changes sides and colors for the teams, and relocates the players and the ball to the predefined positions, depending on the kick-off team. This also takes place when a goal is scored. Starting positions of the robots are shown in Figure 2.11. In the current instance, the blue team is the kick-off team and always defends the blue goal, whereas the red team always defends the yellow goal. When the ball goes outside the field's boundaries, it is returned to a certain position near the exit point. There are two major states in the game; normal play and penalty shootout. Normal play refers to the course of the two periods of play and, if the game ends in draw, then the penalty shootout takes place, where the teams take turns and shoot penalties to decide the winner. Other

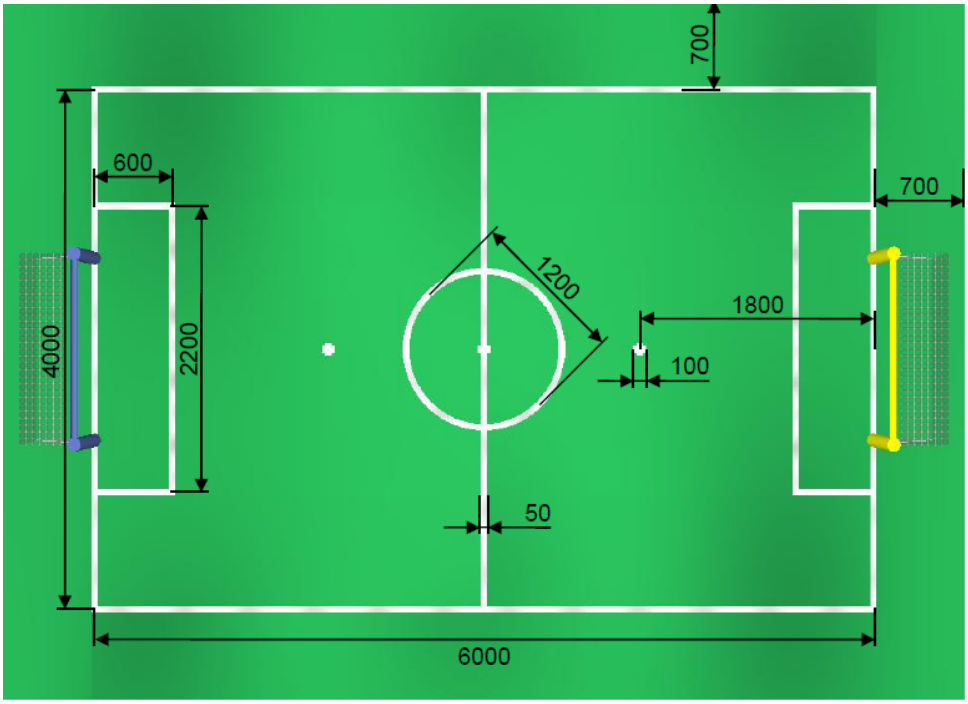

Figure 2.10: SPL 2011 and RobotStadium 2011 field specifications.

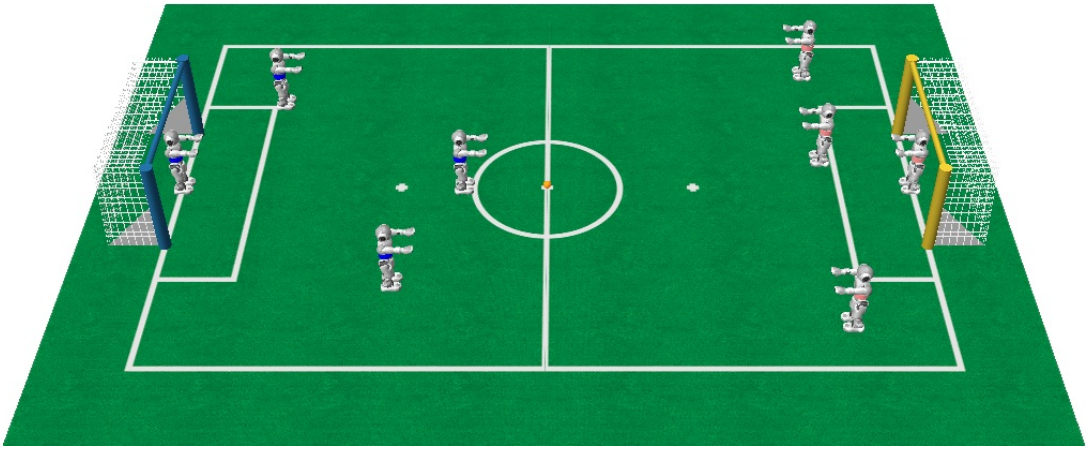

Figure 2.11: Landmarks and kick-off positions.

game states are used at the beginning of the match in the following order: initial, ready, set; when the match ends, the supervisor switches to state finish.

2.3 Mathematical background

Our method for landmark recognition and self-localization uses several mathematical techniques. Here they are going to be presented and explained and their use will be revealed in subsequent chapters.

2.3.1 Least-Squares Fit

The method of least squares [10] is a mathematical procedure for finding the best-fitting curve to an unknown function, when only a finite set of points (values) of the function is known. It determines the form of an unknown equation that best describes our data and therefore the most important application is in data fitting. It does so by solving an undetermined system, i.e. sets of equations where there are more equations than unknowns. The solution minimizes the sum of the squares of the residuals (differences) between the fitting curve and the data points. We will examine two parametric models of curve equations used in least-squares problems:

- $y = ax + b$ (linear)
- $y = c_0 + c_1x + c_2x^2$ (polynomial of degree 2)

where x is an independent variable, y is a dependent variable and a, b, c_0, c_1, and c_2 are the parameters of the model. The goal of least squares is the determination of these parameters. Each parameter describes a different aspect of our curve's behavior. In the polynomial model $c_2x^2 + c_1x + c_0 = y$, parameter c_2 determines the curvature, c_1 the slope, and c_0 the shifting of the curve. Therefore, if c_2 is 0, or nearly 0, then our model describes a straight line. Suppose that the data points are (x_1, y_1), (x_2, y_2), ..., (x_n, y_n). Then the linear system we have to solve in order to find the best parameters of our curve equation is:

$$\begin{pmatrix} \sum_{i=1}^{n} x_i^4 & \sum_{i=1}^{n} x_i^3 & \sum_{i=1}^{n} x_i^2 \\ \sum_{i=1}^{n} x_i^3 & \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i \\ \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i & n \end{pmatrix} \begin{pmatrix} c_2 \\ c_1 \\ c_0 \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{n} x_i^2 y_i \\ \sum_{i=1}^{n} x_i y_i \\ \sum_{i=1}^{n} y_i \end{pmatrix}$$

Figure 2.12 shows examples of least-squares data fitting.

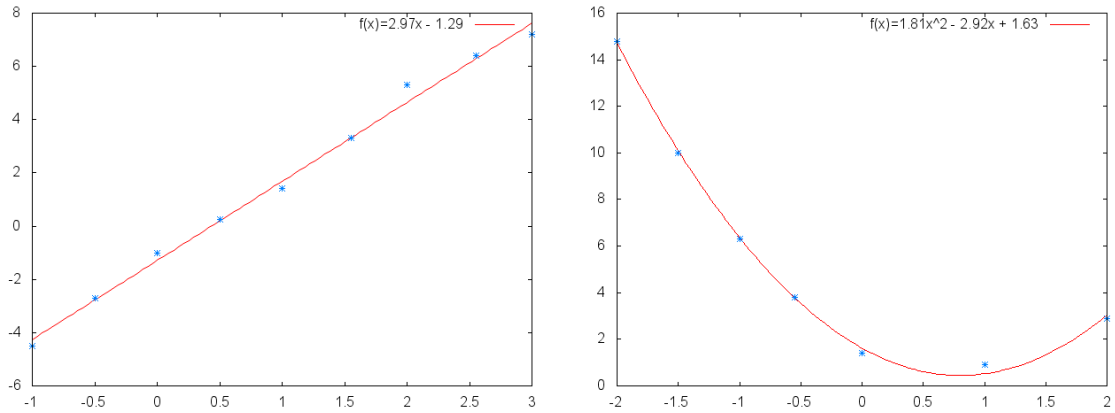

Figure 2.12: Linear (left) and polynomial (right) least-squares fitting.

2.3.2 Fitting a Circle to Three Points

Given three points A, B, C on the plane, this method [11] is used to determine the center point and the radius of a circle that passes through all three points (Figure 2.13). Let us assume that the three points' coordinates are $A(x_A, y_A), B(x_B, y_B), C(x_C, y_C)$. We can form two lines which pass from our given points; line l_1 passes from A and B, and line l_2 passes from B and C. These two lines have the following equations:

$$y_1 = m_1(x - x_A) + y_A$$
$$y_2 = m_2(x - x_B) + y_B$$

where m_1, m_2 are the slopes of the two lines and they are given by:

$$m_1 = \frac{y_B - y_A}{x_B - x_A}$$
$$m_2 = \frac{y_C - y_B}{x_C - x_B}$$

Then, we can form two more lines that pass through the midpoints of line segments AB and BC and are perpendicular to them. The intersection point of these two lines will be the center point M of the circle. Their equations are:

$$y_{PM} = -\frac{1}{m_1}\left(x - \frac{x_A + x_B}{2}\right) + \frac{y_A + y_B}{2}$$
$$y_{QM} = -\frac{1}{m_2}\left(x - \frac{x_B + x_C}{2}\right) + \frac{y_B + y_C}{2}$$

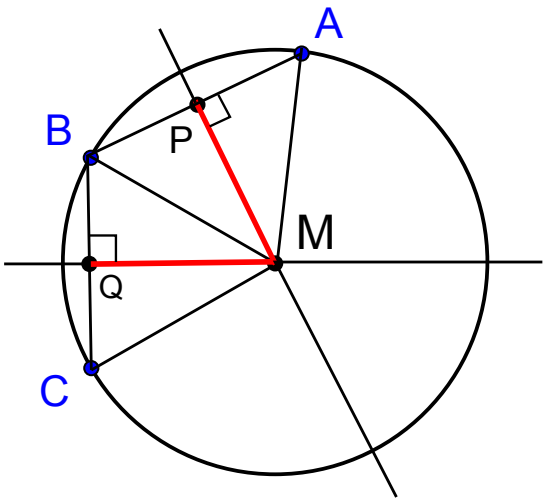

Figure 2.13: Points A,B,C,M and lines QM and PM.

Now, we can find the coordinates of the center point M:

$$x_M = \frac{m_1 m_2 (y_A - y_C) + m_2 (x_A + x_B) - m_1 (x_B + x_C)}{2(m_2 - m_1)}$$

$$y_M = -\frac{1}{m_2}\left(x_M - \frac{x_B + x_C}{2}\right) + \frac{y_B + y_C}{2}$$

In order to find y_M we simply substitute x_M into one of the equations of the perpendicular lines and solve for y. The radius of the circle is equal to the euclidean distance of point M to any of the given points A, B, C. For better understanding of the solution, Figure 2.13 illustrates the aforementioned points and lines.

2.3.3 Intersection Points of Two Circles

When two circles on a plane overlap, at most two intersection points are formed. The following method [12] describes how to find those points using Figure 2.14 as reference. The intersection points are the points $P_3(x_3, y_3)$. The other points' coordinates will be referred to as $P_0(x_0, y_0)$, $P_1(x_1, y_1)$, $P_2(x_2, y_2)$. In order to be sure that the two circles intersect, we calculate the euclidean distance d between the centers of the circles:

$$d = \|P_1 - P_0\|$$

- If $d > r_0 + r_1$, then the circles do not overlap and there are no intersecting points.

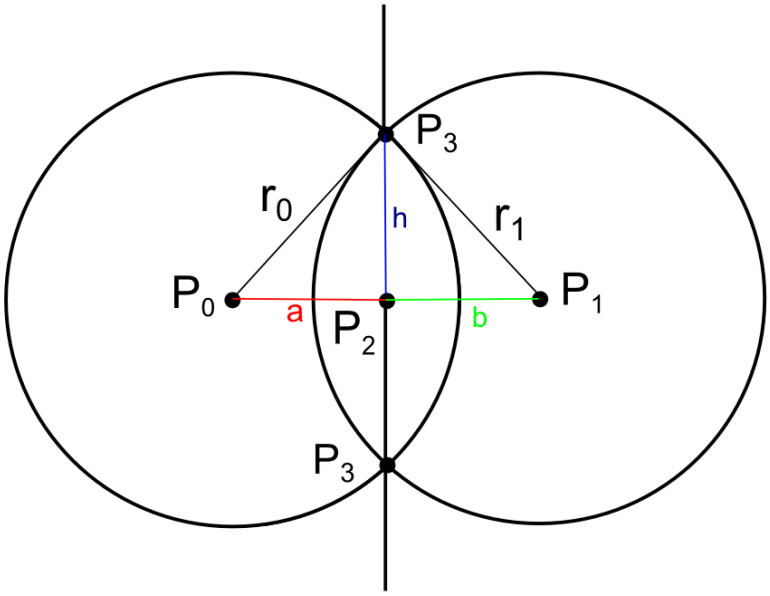

Figure 2.14: Intersection of two circles.

- If $d < |r_0 - r_1|$, then the circles are contained within each other and again there are no intersecting points.
- If $d = 0$ and $r_1 = r_0$, then the circles are coincident and there are infinite solutions.
- If $d = r_1 + r_2$ or $d = |r_1 - r_2|$, then the circles are tangent and there is only one intersecting point which lies along the line connecting the two centers.

If none of the above apply, then we can calculate the two intersecting points. Our first goal is to find h and P_2. We notice that two triangles are formed through points $P_0P_2P_3$ and $P_1P_2P_3$. Thus, from the Pythagorean theorem, we know that:

$$r_0^2 = h^2 + a^2$$
$$r_1^2 = h^2 + b^2$$
$$d = a + b$$

Considering the above equations, we can solve for a,

$$a = \frac{r_0^2 - r_1^2 + d^2}{2d}$$

Now we can find h by substituting a into equation $h^2 = r_0^2 - a^2$ and P_2 with the equations:

$$x_2 = x_0 + \frac{a(x_1 - x_0)}{d}$$
$$y_2 = y_0 + \frac{a(y_1 - y_0)}{d}$$

It remains to calculate the two sets of coordinates for the two P_3 points:

$$x_3 = x_2 \pm \frac{h(y_1 - y_0)}{d}$$
$$y_3 = y_2 \mp \frac{h(x_1 - x_0)}{d}$$

Chapter 3

Problem Statement

In this chapter we state the problem we study in this thesis, namely visual recognition of the static landmarks in the RobotStadium field (goals and lines) and self-localization of the robots based on the recognized landmarks. To understand better the challenges behind this problem, we first provide a detailed description of the camera sensor and the images it delivers.

3.1 Simulated Nao Camera

The resolution of the simulated Nao camera is 160 pixels in width and 120 in height. It has a limited horizontal field of view of 46° and a vertical field of view of 34°, as shown in Figure 3.1 and Figure 3.2 respectively. In comparison to the real Nao camera, it has a lot less noise and the colors are more distinguishable. Figure 3.3 shows an example camera image, whereby the player is standing still at 3.6m from the goal near the center of the field. The image suffers by a bit of distortion, when the simulated robot moves, but this is done by the simulator in an attempt to resemble the distortion of the real camera. The horizontal position index of the image has a range of $[0, 159]$ and the vertical position index has a range of $[0, 119]$. The $(0, 0)$ position is located at the upper left corner of the image.

There are however some drawbacks in the simulated camera, that renders the image processing harder than it looks:

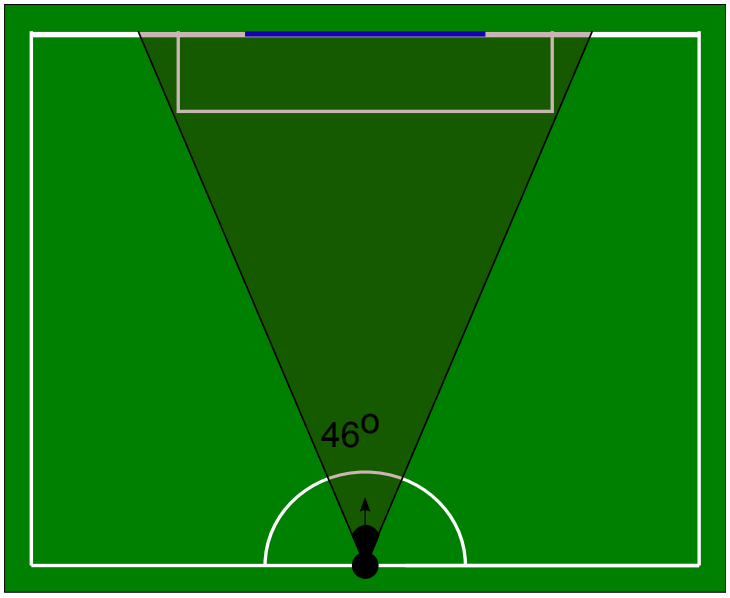

Figure 3.1: Simulated Nao camera horizontal field of view.

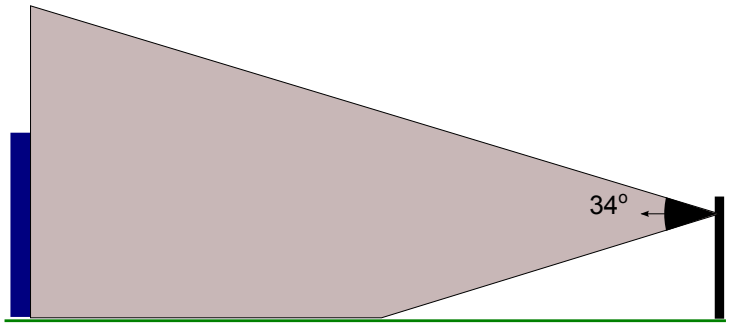

Figure 3.2: Simulated Nao camera vertical field of view.

- The small resolution reduces dramatically the camera's capabilities to recognize an object, when it is rather far in the field. The width of the goal posts is sometimes presented less than it should, lines disappear or get merged and, even if they're visible, they are only one pixel thick. Figure 3.4 presents examples of this problem. The two images represent two consecutive frames captured as the player was in motion at about 3.5m from the goal. The field base line is not visible in the left image, whereas the main goal area line is not visible in the right image. Also notice that the goal posts in the left image have more width than those in the right image, even though the images were taken from the same position.

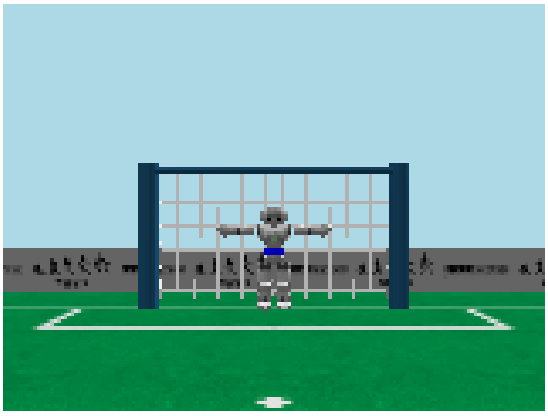

Figure 3.3: Simulated Nao camera image example.

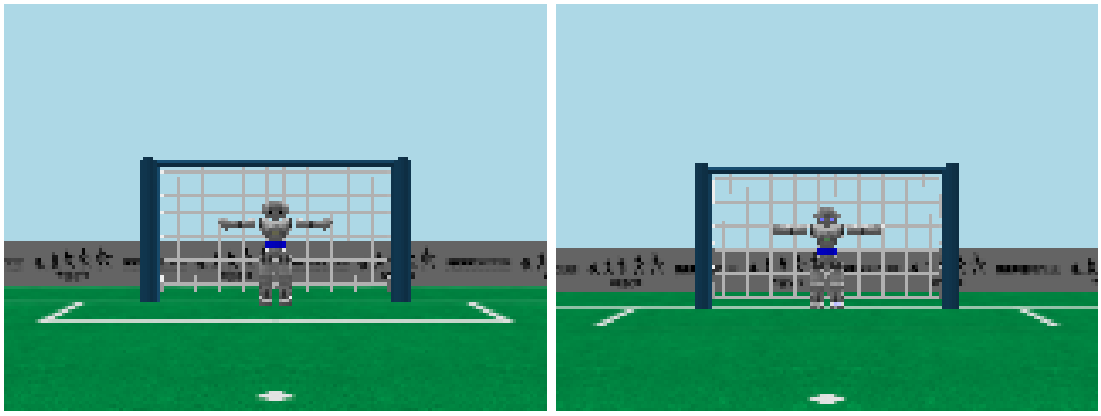

Figure 3.4: Camera image examples with missing lines.

- The small pixel count means that the pixel-to-pixel difference on the image has quite a significant impact in estimating the direction and the distance of some object in the field. So, if we want to obtain accurate estimations, we have to be very precise about the positions of the pixels corresponding to the object of interest.

- The limited field of view forces the robot to scan the field by panning its head and executing the landmark recognition procedure several times in order to collect enough information for self-localization.

In general, the camera image provides information, however with significant errors. So, in the interest of making our image data dependable, a lot of effort is required to deal with this problem during the image processing procedure.

Figure 3.5: Camera images containing a goal from different views.

3.2 Landmark Recognition Problem

In robotic football, simulated or not, implementing landmark recognition through vision is indispensable. It enables the player to know at any given instance the kinds of object that fall within the visual field of the camera. Accurate direction and distance estimates of the visible objects are essential not only for self-localization, but also for the player's decision making algorithm and behavior in a game. In general, landmark recognition provides the agent with the ability to perceive its surroundings in any given situation, rendering it today a fundamental step in a robot's development.

For the purposes of this thesis, specific landmarks of the football field were recognized in order to achieve our objective, which is the self-localization of the player. These are the objects that have a fixed position in the field, for example the ball cannot be perceived as a field landmark.

3.2.1 Goals

The two goals are the most helpful landmarks in the field. Being one blue and the other yellow, the two goals are easily distinguishable, as shown in Figure 3.5. The robot's vision has to detect blue or yellow pixels to separate the own from the opponent goal. The only problem encountered, other than errors in measurements, is that in the occasion where only one goal post is visible (Figure 3.6), it is essential to be identified as left or right. This is necessary, because posts are treated as individual landmarks in the field and a player's distance from the left post usually differs from its distance from the right post.

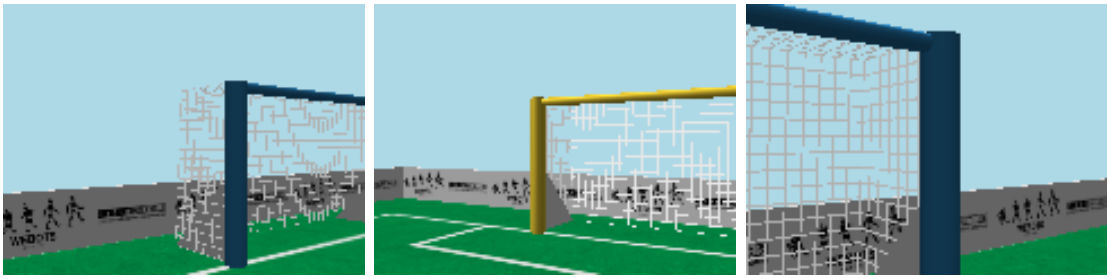

Figure 3.6: Camera images containing only one post of a goal.

3.2.2 Field Lines

Unlike goal recognition, recognition of field lines is a more complex task. White pixels, which may indicate the presence of a line, can be found anywhere in the robot's camera image, mostly due to other objects, such as other players or banners outside the field, which occasionally are sensed as white in the image. Therefore, areas of white pixels that indicate lines must be isolated from the rest. Finding the best settings in order to separate the desired white areas as efficiently as possible is pretty challenging due to the camera's drawbacks mentioned above and the unlimited number of game situations that may occur and projected on the camera. This is true, because the environment in which our agent has to act is dynamic due to the presence of multiple agents. For example, another agent may block the field of view of our agent or blend with field lines in the image, but in any case it alters entirely the way we acquire the information we need.

Even if a line segment has been correctly recognized, it is still necessary to characterize it as a straight line, a curved line, or a corner line. A problem here that has to be surpassed is how to differentiate between curved lines and corner lines. In order to do that, lines may have to be split and merged depending on the occasion. The main use of line recognition is the determination of the following key landmarks:

- **Center circle** This is a very useful landmark, due to its positioning and uniqueness. Being in the middle point between the two goals, its recognition is imperative for self-localization for it covers a large area of the field, where there are no other unique landmarks. A specific problem is that the center circle is projected on the camera as an ellipse. Figure 3.7 shows how the center circle is viewed from different angles.

Figure 3.7: Wholly-visible (left) and partially-visible (center and right) center circle.

- **Type L corners** There are eight type L corners in the field; four at the field edges and four at the goal areas. The main characteristic is the change of direction in the image depending on the angle of view. Figure 3.8 shows how various corners in the field are viewed.

- **Type T intersections** Although there are six type T intersections, only two of them are really useful, the ones at the end of the middle line. The intersections next to the goal posts can be ignored due to existence of better landmarks nearby. Figure 3.9 shows how several type T intersections are viewed.

Random straight line segments could also be detected, but they cannot offer useful information because of the multitude of their possible positions.

3.3 Self-Localization Problem

Self-localization as a problem is crucial in order to achieve autonomy for an agent. If a robot has conviction about its own position, then it is easier to make decisions regarding future actions. But to do so, the agent needs information capable of providing the means to find where it is. This information can be obtained by sensors (e.g. camera, laser, sonar) or by an odometer. There is always an uncertainty in measurements, so consequently faults in the estimated position are expected. The goal of localization is to determine the own position as accurately as possible.

Several techniques, such as Kalman filters, particle filters, and constraint-based methods, are used to achieve localization. In robotic football, localization gives a significant advantage to a player. It is widely addressed by most RoboCup teams. Game strategy,

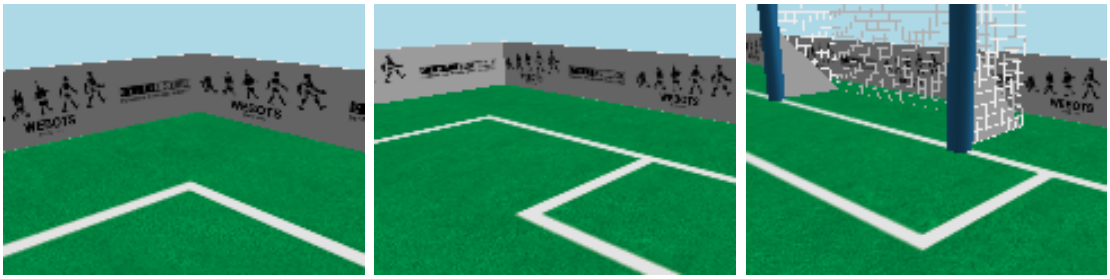

Figure 3.8: Type L corners viewed from various angles.

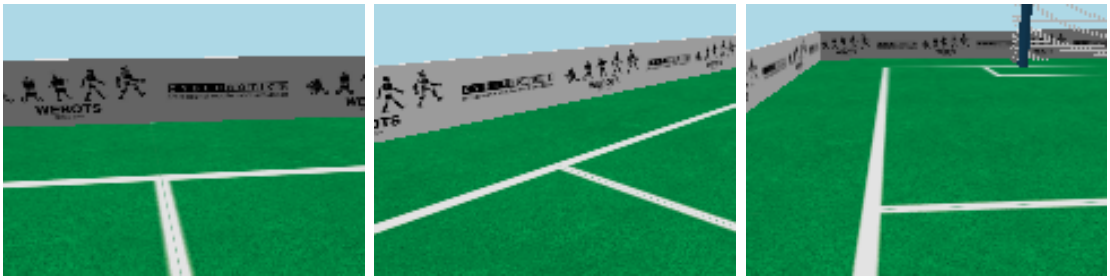

Figure 3.9: Type T intersections viewed from various angles.

team cooperation, and player movements can be guided through position estimations provided by a localization method. A player's field position is characterized by three variables; x, y, and θ, where (x, y) are the coordinates on the field plane and θ is the angle value that describes the player's orientation.

The primary problem that has to be confronted in localization is how to deal with the errors present in the estimations. Landmark recognition is not carried out perfectly and small deviations of the landmarks' estimated positions in regard to the player are expected. However, even small, those deviations can become significant during the computation of the candidate positions of a player in the field, so a filtering procedure should be implemented that chooses the best candidate. Besides this, a landmark with serious error measurements may not be taken into consideration for the self-localization process.

Chapter 4

Landmark Recognition

In this chapter we will explain how the camera image processing was done, with the purpose of recognizing the following field landmarks:

- 4 goal posts
- 2 goal centers
- 1 field center circle
- 4 L type junctions at the field corners
- 4 L type junctions at goal areas corners
- 2 T type junctions at the ends of the middle line

This is a total of 17 landmarks. During the process of scanning for landmarks, the robot's head ($HeadPitchAngle$) is turned 17° downwards and the top position of the camera is used, so as to have the best view over the field as the head pans left and right. In the present chapter, when coordinates are mentioned, the intention is to pinpoint a pixel's position on the image.

Our landmark recognition module was implemented to aid only the self-localization process of the player, therefore ball recognition is not covered. Ball detection during game is served through a different recognition procedure.

Color	R	G	B
	255	0	0
	0	255	0
	0	0	255
	255	255	0
	0	255	255
	255	0	255
	255	140	0
	0	0	0
	255	255	255

Table 4.1: RGB values of used colors.

4.1 Color Segmentation

The image given by the simulator is an array with a length that corresponds to the number of pixels in the image. Each element of this array represents one pixel coded in RGB (Red, Green, Blue) model, whose levels can be accessed. Every pixel of an RGB image is actually a vector consisting of three values; one for each of the Red, Green, and Blue components in the range of $[0, 255]$, if we refer to an 8-bit color depth image. Every combination of these three values produces a different color. Basic colors are Red, Green, and Blue, and secondary colors are Yellow, Cyan, and Magenta. All of these colors were used in our work in addition to Black, White, and Orange. Table 4.1 illustrates their RGB values.

The color segmentation of the camera image is imperative. It divides all pixels into a certain subset of colors making image processing and landmark recognition a lot easier. The red, green, and blue values of each pixel are provided to a simple algorithm which uses thresholds and color priorities, depending on the percentage of every color in the field, to determine its color. Simply put, every pixel in the image is examined if it fits into a particular range of values, depending on the color we want to fit it with. If the pixel does not qualify for the first color, then we examine the next one, and so on. The color priority sequence is: green, white, blue, yellow, orange, and cyan. If the algorithm fails to fit a pixel into a specific color, the pixel becomes white. Some faulty outlier pixels may

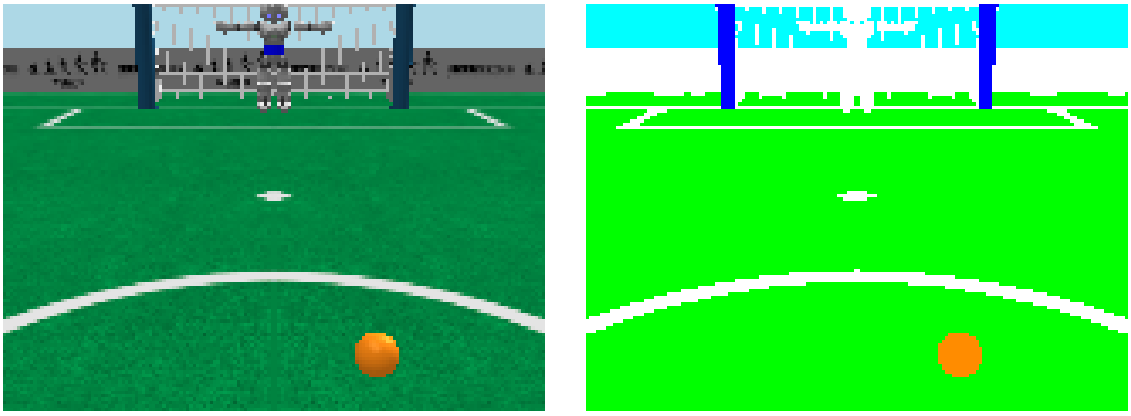

Figure 4.1: Camera raw image (left) and color-segmented image (right).

appear, but it takes little effort to fix, by checking the colors of the neighboring pixels. When a pixel is identified as a color other than white and there are no neighboring pixels with the same identified color, then the pixel will be taken as white. If, however, a pixel is identified as white and there are no other neighboring white pixels, then it will be taken as green or orange, depending on the existence and the number of neighboring orange pixels. Figure 4.1 offers an example of color segmentation.

4.2 Direction and Distance Estimation

For each recognized landmark, we need to be able to know its exact direction angle and distance in regard to the player's position. Every pixel on the image represents a different segment of the visible area of the field. Consequently, we can extract the distance and direction to that field segment from the pixel's coordinates. Our convention for angles is that positive angles are to the right or to the top (clockwise), whereas negative angles are to the left or to the bottom (counter-clockwise).

To determine direction, first we have to find the direction angle between the camera center and the target pixel on the image using the following equation:

$$DirectionAngle = \left(\frac{w}{width} - 0.5\right) \times horizontalFOV$$

where w is the pixel's width (horizontal) coordinate and $width$ is the width of the image. If the robot's head is looking straight forward, then the $DirectionAngle$ we estimated is the direction in radians we are looking for. However, when the head is turned in any

way, we have to add the horizontal turn angle of the head to our estimation, assuming it follows our convention for angles[1]:

$$Direction = DirectionAngle + HeadHorizontalAngle$$

Before we can calculate the distance, the elevation angle between the camera center and the target pixel on the image has to be estimated:

$$ElevationAngle = -\left(\frac{h}{height} - 0.5\right) \times verticalFOV$$

where h is the pixel's height (vertical) coordinate and $height$ is the height of the image. The elevation angle is only affected by the pixel's height coordinate, hence pixels with the same height coordinate all have the same elevation angle. Much alike direction, we have to add the vertical turn angle of the head and the camera offset angle to our estimation, assuming that both of them follow our convention for angles:

$$Elevation = ElevationAngle + HeadVerticalAngle + CameraOffsetAngle$$

Possessing the above information we can now compute the distance between the player and the segment of the field corresponding to the target pixel (target segment). First, we compute the distance corresponding to the elevation of the target pixel assuming a zero direction (zero segment):

$$y = \frac{0.51}{\tan(-Elevation)}$$

where 0.51 is the robot camera height with respect to the ground. Figure 4.2 presents an example of estimating the distance y to the zero segment, when the $HeadVerticalAngle$ and the $CameraOffsetAngle$ are both zero, and therefore the $Elevation$ is equal to the $ElevationAngle$. Then, we compute the distance between the target and zero segments:

$$x = y \times \tan(-DirectionAngle)$$

Finally, using the Pythagorean theorem we estimate the distance from the player to the target segment:

$$d = \sqrt{x^2 + y^2}$$

Figure 4.3 presents an example of the above procedure for estimating the distance to the base of the blue right post.

[1] In RobotStadium, the angle values for the robot joints follow the exact opposite convention.

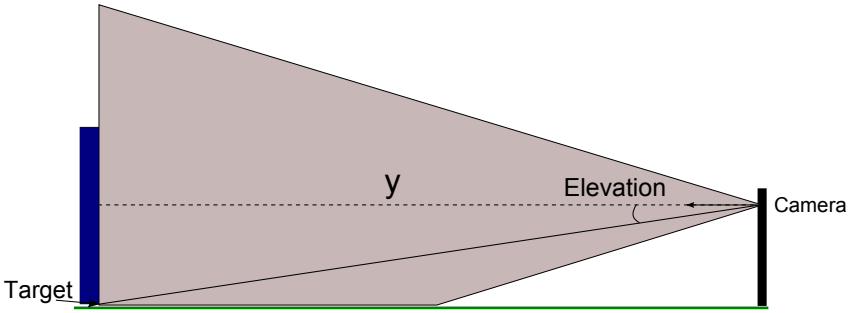

Figure 4.2: Estimation of the distance corresponding to the elevation of a target pixel.

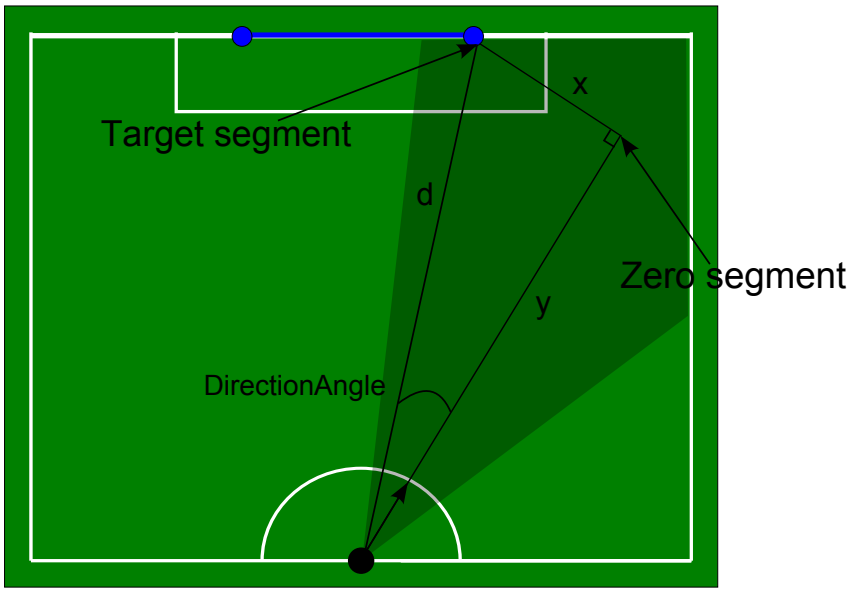

Figure 4.3: Distance measure example.

4.3 Field Border Detection

Finding where the field ends is very useful for the detection of lines in the field. It reduces the number of pixels that have to be processed because the field border acts as a limit. Furthermore, the detection of a field border in the image determines whether the rest of the landmark recognition procedure will continue. Under normal conditions and due to our choice for the *HeadPitchAngle* during landmark recognition, the field border must be visible. If it is not visible, it is an indication that the player may have fallen on the ground or his camera has been obstructed by other players. In those instances, no landmarks are visible, the field border cannot be detected and, therefore, the early

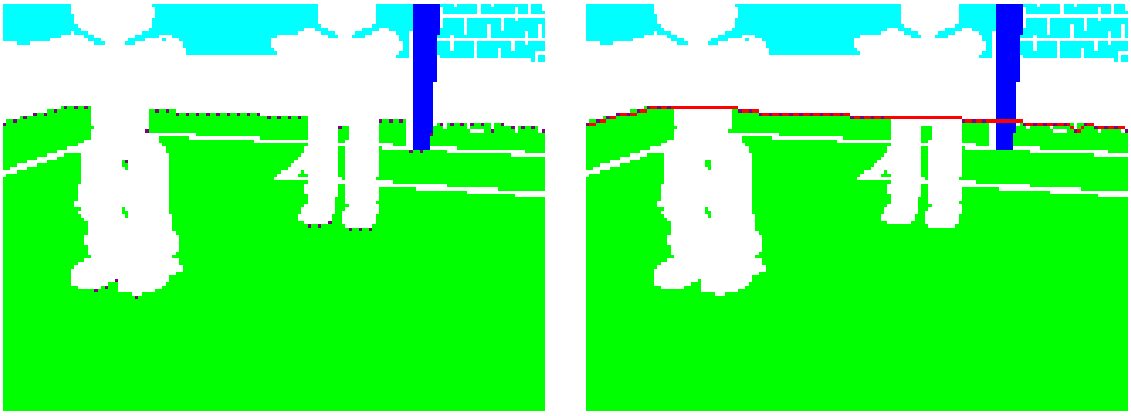

Figure 4.4: Field border detection: before and after the Find Border Line call.

termination of landmark recognition saves the player from unnecessary image processing.

Field border detection (Figure 4.4) begins with finding the first green pixel and marking it, starting from the top of the image and going downwards. This is done every third column over the entire image, saving the image height coordinates of the marked positions (shown as dots in Figure 4.4). Then, a function is called to determine whether a field border line can be drawn on the image along these dots or not. This function, shown in Algorithm 1, takes as inputs the array with the height coordinates, a starting point, a pixel height threshold, and a rejection percentage and returns a boolean variable, indicating whether a field border line was detected or not, and an array containing the field border line. Using the height coordinates and the pixel height threshold, it attempts to connect the dots. If a dot exceeds the limit (pixel height threshold) with respect to the previous dot in the border line, then it is ignored (it is marked with a -1) and the loop continues with the next dot. This is done in order to avoid including obstacles above or below the field border line. During the loop, an assessment is made, depending on the number of ignored dots, to determine early if the field border line cannot be drawn and has to be rejected; in this case, the function simply returns $False$. If a sufficient number of dots are successfully connected, the function returns $True$ along with an array containing the heights belonging to the detected field border line (the detected border line is shown in red in Figure 4.4). There is a symmetric function that detects a field border line starting from the right side of the image. In order to exhaust all possibilities, these two functions alternate as long as they return $False$ with starting points that begin at the extremes of the image and proceed towards the middle until one returns $True$.

Algorithm 1 Find Field Border Line from the Left Side of the Image.

Input: $ArrayHeight$, $Start$, $HeightThreshold$ (default value = 3), $RejectionPercentage$ (default value = 0.75)
Output: $FoundLine, BorderLine$
$BorderLine = ArrayHeight$
$Left = Start$
$Right = Start + 1$
$Counter = 0$
while $Right < size(ArrayHeight) - 1$ **do**
  if $BorderLine[Left] == -1$ **then**
    $Left++$
  else
    if $\left|ArrayHeight[Left] - ArrayHeight[Right]\right| < HeightThreshold$ **then**
      $Left++$
    else
      $BorderLine[Right] = -1$
      $Counter++$
    end if
    $Right++$
  end if
  if $Counter > RejectionPercentage \times size(BorderLine)$ **then**
    return $False$
  end if
end while
return $True, BorderLine$

4.4 Goal Recognition

Goalposts are the most characteristic landmarks in the field. Their size and color uniqueness make them easy to detect and very important during the localization procedure. As mentioned at the beginning of this chapter, our robot's head is turned a bit downwards to achieve maximum view of the field. So only the vertical posts of the goals can be seen in the camera image, but that is sufficient for goal detection. The entire image is scanned vertically and when a pixel is identified as part of a goal (due to its color), its

coordinates are stored to examine them further as an object and not as a single point. The goal examination includes the following basic steps:

1. First, we need to determine if both posts are visible or just one. The stored coordinates are checked for a gap in width (the default value is 20 pixels) and in that occasion we split the pixels into two different post objects.

2. Next, we have to locate the pixel corresponding to the base of each post, so that we can use its coordinates to estimate direction and distance. This is done by finding the average width of the pixel concentration belonging to the post on the image and the maximum value of height (lowest in the image) among these pixels.

3. When both posts are detected, we find two base pixels, one for the left post and one for the right post. In this case it is very useful to determine the goal center position to treat it as an extra landmark. Average values of base pixels are calculated to do so.

4. If only one post is visible then the problem of identifying which one was detected is solved by turning the robot's head upwards, until the horizontal post of the goal is found. Depending on the direction of the horizontal post (left or right of detected vertical post), we decide which post was initially found.

Sometimes during the game, a post can be obstructed by another robot and the pixel concentration that we detect is not sufficient to make a proper assessment. To avoid using wrong measures and to verify our findings, two checks are being performed:

1. If both posts are visible, then we compare their heights (difference between maximum and minimum height coordinates in the concentration) on the image. If their height difference is greater than a given threshold (default value is 40 pixels), then the post with the lower height is rejected.

2. If a post's distance estimate is longer than our field's dimensions, then it is rejected.

In the above cases, due to the lack of both post objects, the calculation of the goal center does not take place. In Figure 4.5 we can see examples of goal recognition; goal posts are outlined with a purple border.

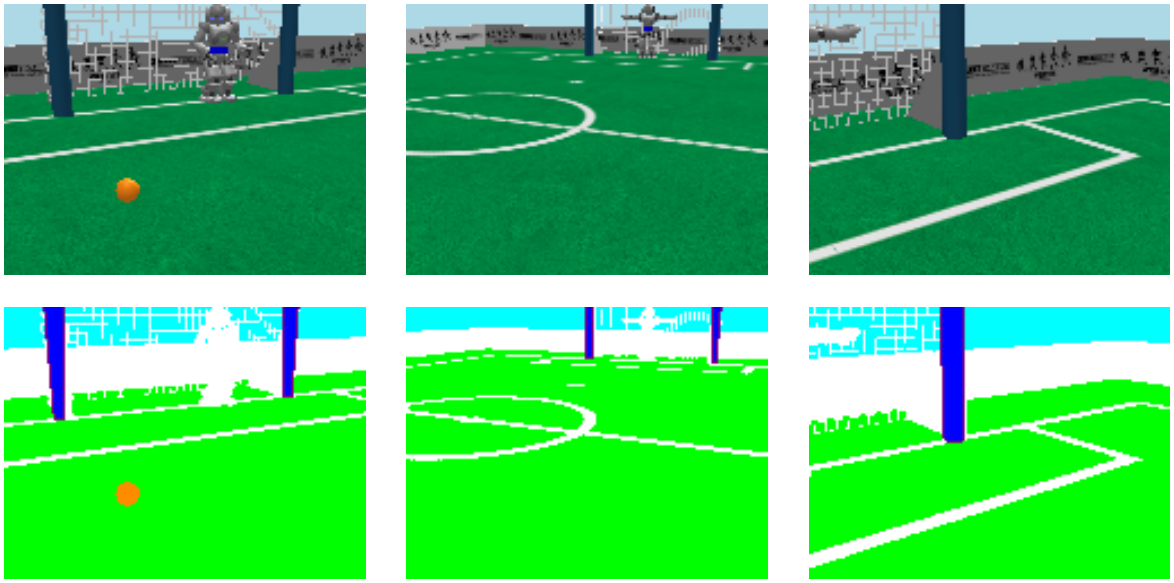

Figure 4.5: Goal recognition from various distances and angles.

4.5 Field Lines Recognition

4.5.1 Scan Lines

For the white areas of the image to be marked, we run a vertical scan that starts from the field border up to the bottom of the image every fifth column. These five pixels define the constant width difference between any two scan lines and hereafter will be mentioned as *dist*. Every white pixel found is turned black, thus creating small black lines on white segments all around the segmented image. Figure 4.6 illustrates the use of scan lines. Black lines indicate the possibility of a field line. Notice that in some cases a robot's body may cover a considerable percentage of the white areas.

Some of the black lines are part of the field lines we want to detect and some are robot's parts. In order to include the former in our recognition process and reject the latter, certain criteria had to be matched for every black line. The criteria that are used in the matching process are the following:

Criterion 1 All black lines that have as a starting or ending point a pixel other than green are being discarded. For example, if a black line starts from the field border line then it has to be rejected. Also this criterion is a precaution for any faults in the color segmentation sequence.

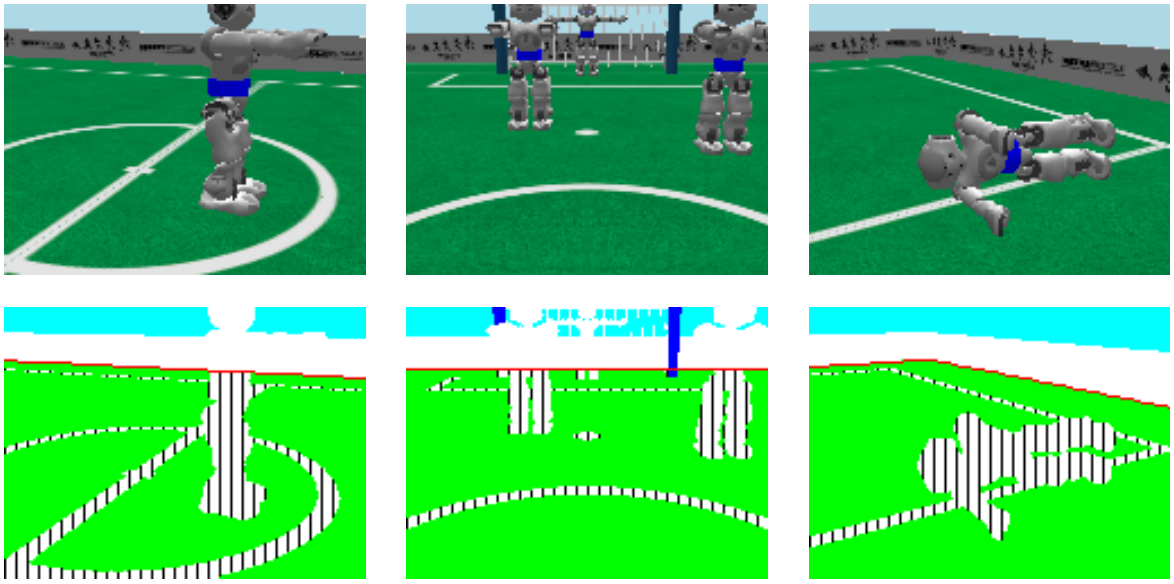

Figure 4.6: Vertical scan lines below the field border for field line indication.

Criterion 2 If a black line has other than white pixels next to its interior points (the two ends are ignored), then it is being rejected.

Criterion 3 Black lines must not be longer than a particular pixel count threshold, which is calculated depending on the average length of all black lines on the image that are larger than three pixels. The threshold is twice this average.

Criterion 4 For a black line to be a part of a line segment, it has to fit into a certain continuity of black lines. Therefore, a procedure is responsible to determine for each black line if it belongs to a series of three "close-enough" black lines. By "close-enough" we mean that there should be some kind of overlap in height coordinates between neighboring black lines. Any black line that fails to meet these conditions is rejected. Furthermore, this criterion discards any black line that is significantly smaller compared to the next black line.

As seen in Figure 4.7 the above criteria can minimize the number of undesirable white areas in the recognition process. Each black line that is erased by a different criterion is painted with a specific color. Criterion 1 is orange, 2 is magenta, 3 is cyan, and 4 is yellow. Those that remain black will be further used for recognition. Nevertheless, due to the fact that we are dealing with a dynamic environment faults are expected. Some

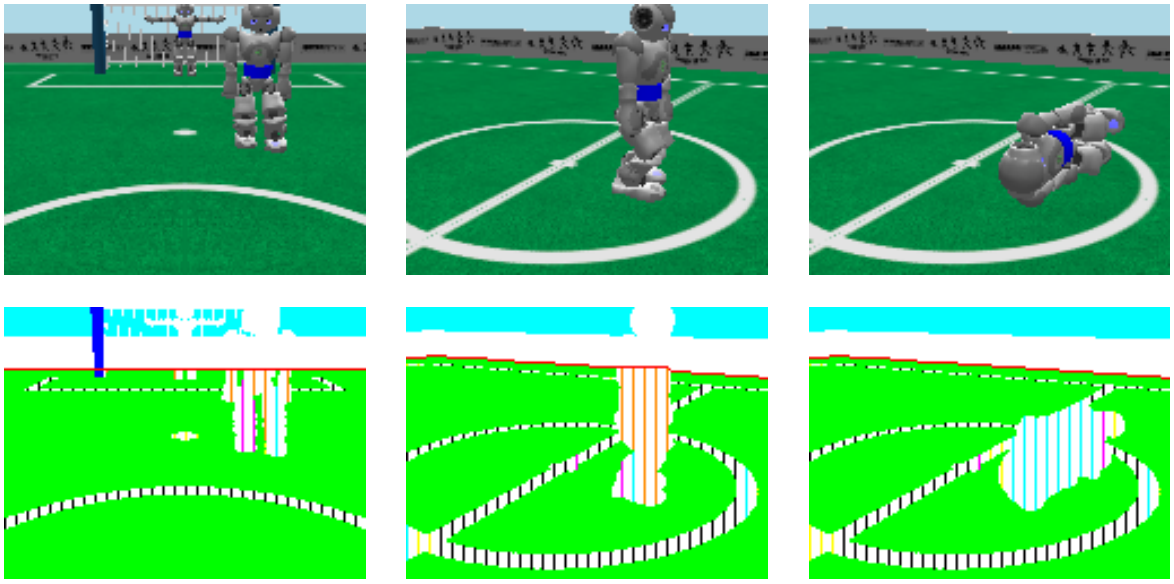

Figure 4.7: Application of the four rejection criteria on black lines.

unwanted black lines will pass the criteria and some that are desirable will be sacrificed. Our goal was to distinguish them as efficiently as possible, so the existing errors can be taken into consideration and surpassed at a succeeding recognition stage.

4.5.2 Line Formation

After the scan lines have been filtered we need to determine our visible lines. The valid black lines from the previous subsection viewed as a sequence form the lines we want to detect. The *AddLines* function performs the process of finding those sequences by taking into consideration various parameters we are about to analyze. *AddLines* embodies three other functions: *GetStart*, *GetLine*, and *Split*. Here, we will explain the first two functions and the last will be explained in the next subsection:

- **GetStart** All valid black lines attributes (coordinates on image, length, middle pixel) are stored vertically beginning with the one that is upper left in the image and ending with the one that is lower right in the image. This function's aim is to find the starting point of a field line by iterating through the black lines array. It starts by marking the first black line as the start of the first field line. If it is called again it searches to find the next available black line that can be a possible start of

a field line. When all black lines have been used and therefore there are no more starting points, the function does nothing and -1 is returned.

- **GetLine** This function takes as input the starting point from *GetStart* and iterates through the black lines array in order to find and add those that belong to the particular field line. It compares the attributes of the current black line (`current`) with those of the next one in the array (`next`), so that all following criteria are met:

  1. `next`'s flag attribute has to be 0, meaning that it wasn't added to another line.
  2. `next` must not have the same width coordinate as `current`.
  3. The difference in width coordinates between `current` and `next` must not exceed the value of $6 \times dist$. In this occasion, the *GetLine* function is interrupted and the loop does not continue to the next black line. This is done to prevent adding black lines that are very far from each other in the image.
  4. `next` and `current` difference in height coordinates of their midpoints must not exceed a certain limit. This limit depends on their width difference on the image and their average length. If `current` has a length of len_1 pixels and a width coordinate of w_1, and respectively `next` has len_2 and w_2, the limit will be computed as:

     $$AddLimit = \frac{len_1 + len_2}{2} + \frac{w_2 - w_1}{dist} - 1$$

     provided that len_1 and len_2 do not have a large deviation. If they have a large deviation, the criterion is considered as failed, to avoid connecting black lines with highly unequal lengths.
  5. To avoid adding corners to a single line at this stage, the local direction is being tracked using also the previous black line in addition to `current` and `next`. Their height differences are being examined to conclude if direction is valid, i.e. it proceeds without changing sign.

If the detected line is less than three black lines in length, then it is rejected, along with the particular black lines. Line formation examples are shown in Figure 4.8.

Algorithm 2 *GetLine* function.

Input: $BlackLines, start, dist$
Output: $Line$
$Line = Add(\texttt{NULL}, BlackLines[start])$
$current = start$
$next = start + 1$
while $next < BlackLines.size$ **do**
  if $BlackLines[next].width - BlackLines[current].width \geq 6 \times dist$ **then**
    break
  end if
  $AddLimit = getAddLimit(current, next, dist)$
  if $CheckCriteria(BlackLines, current, next, AddLimit) == true$ **then**
    $Line = Add(Line, BlackLines[next])$
    $BlackLines[next].flag = 1$
    $current = next$
  end if
  $next++$
end while
return $Line$

4.5.3 Line Splitting and Merging

At this stage we possess the formed lines each consisting of a set of black lines. Although a local misdirection check (Criterion 4) was carried out in the *GetLine* function, a more extensive direction comparison is performed next along with a procedure that examines if any two lines can be merged. There are two reasons that declare these necessary. First, we don't want our lines to be formed as corners yet, due to the great possibility of misinterpreting a corner as a curve, and secondly, to correct any mistakes made in the previous procedure of line formation. The goal here is to arrange our lines as distinguishable as possible. Thus, with the discovery of a misdirection, the line is split into two parts. The *Split* and *Merge* functions are thoroughly explained:

- **Split** This function examines and tags each detected line with an appropriate direction sequence. Initially, the line is divided into non-overlapping adjacent segments of three black lines each. Each segment is analyzed and characterized depending on

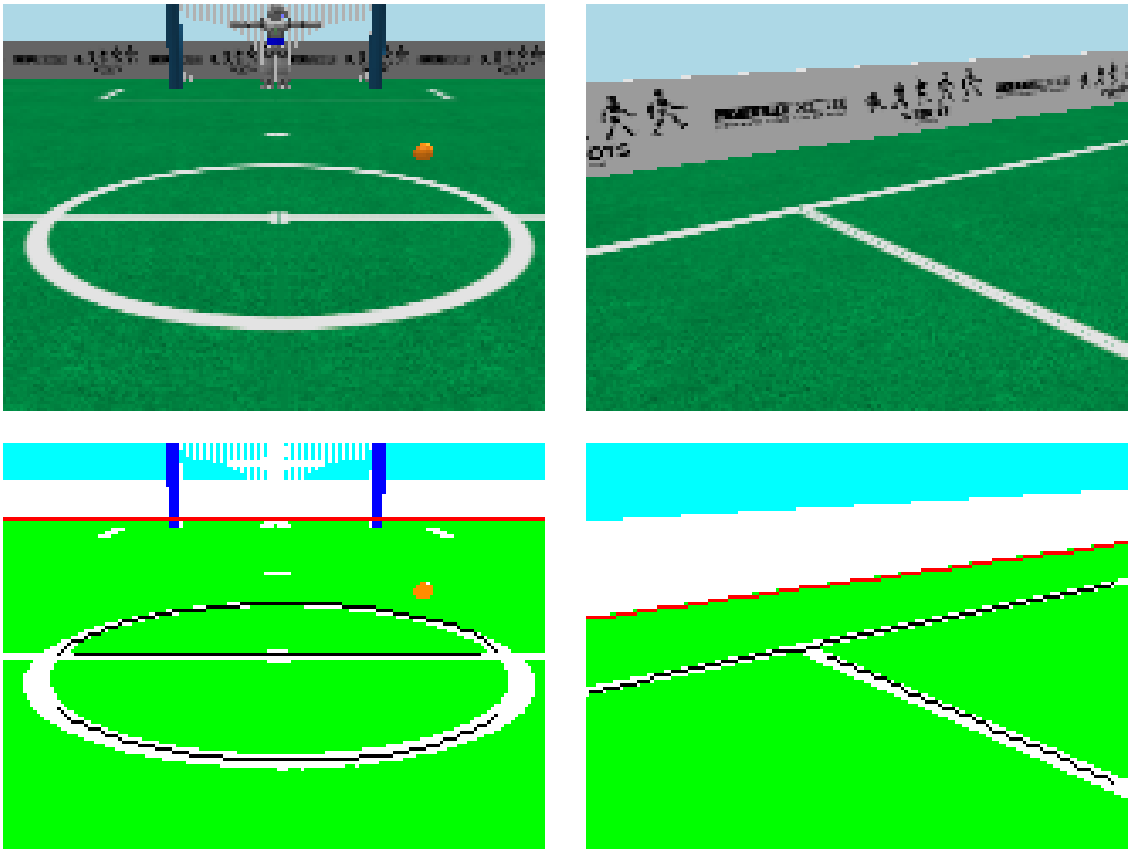

Figure 4.8: Formation of lines after the grouping of valid black lines.

the height differences of the midpoints of the black lines in the segment. Note that the height differences within a segment cannot have opposite signs, because of the last criterion checked by *GetLine*. If the segment has an upwards inclination, then it is labeled *up*, if downwards, *down*, and, if there is no inclination, *straight*. Next, we need to find all direction changes that occur between the segments of a particular line and store them along with the index of the segments in which the direction changes take place. The occurred direction changes describe the line's tendency. For example, if all segments of a line are labeled *up*, then there are no direction changes and the whole line is labeled with only *up*. In contrary, a curved line has direction changes between its segments and, therefore, it will be marked as one of the following sequences: *up*, *straight*, *down* or *down*, *straight*, *up*. The procedure up to this point is described in Algorithm 3. Afterward, we examine the sequence patterns that emerge from each line and, if an unacceptable one is found, then the

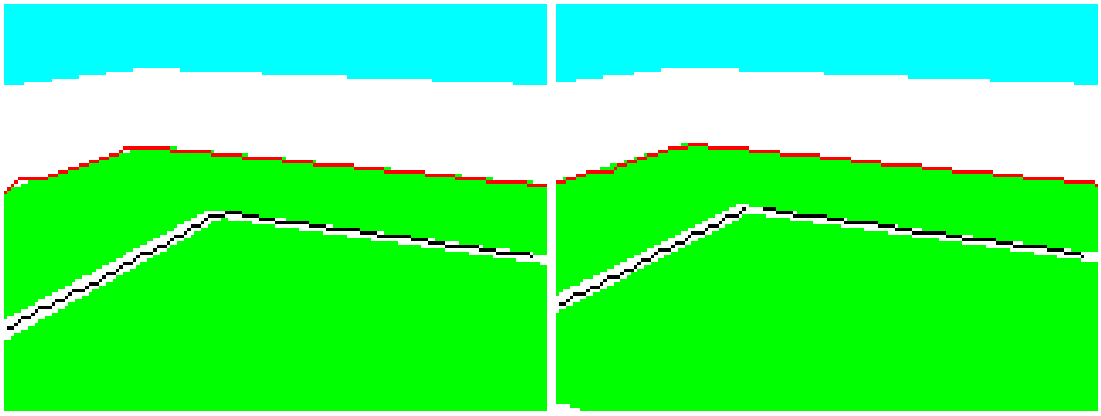

Figure 4.9: Before the use of the *Split* function (left), and after (right).

line gets split at the segment where the direction change took place. Unaccepted sequences are the following:

1. The labels *up* and *down* cannot be consecutive in a line's direction sequence in order to avoid corners. Figure 4.9 illustrates an example of a corner split into two side lines.
2. The sequences *up*, *straight*, *up* and *down*, *straight*, *down* indicate that a side line has been added to a circle line and thus they have to be split.
3. If a *straight* label has been added to the beginning or the end of a circle's direction sequence, then it has to be removed.

Table 4.2 summarizes the possible direction sequences that emerge after the completion of the splitting procedure.

- **Merge** After all black lines have been grouped into lines and split in case of misdirection, a merge procedure is responsible for comparing all pairs of the formed lines and merging those pairs that can fit the following:

  1. Compatibility. Two line directions are compatible if they can be merged into acceptable line forms. Referring to Table 4.2, the list of compatible directions includes only the following ordered pairs: $S_1 - S_1$, $S_2 - S_2$, $S_3 - S_3$, $S_4 - S_2$, $S_6 - S_1$, $S_2 - S_5$, $S_1 - S_7$.

Algorithm 3 Find Direction Labels of a Line.

Input: $Line$
Output: $LineDirection, ChangeSegments$
$segments = SplitLineIntoSegments(Line)$
for all $s \in segments$ **do**
  $diff_1 = s.height_1 - s.height_2$
  $diff_2 = s.height_2 - s.height_3$
  if $diff_1 + diff_2 \geq +2$ **then**
    $s.direction =$ "*up*"
  else if $diff_1 + diff_2 \leq -2$ **then**
    $s.direction =$ "*down*"
  else
    $s.direction =$ "*straight*"
  end if
end for
$currentDirection = s[0].direction$
$LineDirection = Add(\texttt{NULL}, s[0].direction)$
$ChangeSegments = Add(\texttt{NULL}, s[0])$
for all $s \in segments$ **do**
  if $currentDirection \neq s.direction$ **then**
    $currentDirection = s.direction$
    $LineDirection = Add(LineDirection, s.direction)$
    $ChangeSegments = Add(ChangeSegments, s)$
  end if
end for
return $LineDirection, ChangeSegments$

2. No overlapping. Two lines cannot be merged if they overlap in the vertical dimension; this is easily checked using the width coordinates of their ends.

3. Continuity. The last segment of first line and the first segment of the second line must be close enough in terms of width (default: no more than 20 pixels) and height (default: no more than 10 pixels) coordinates.

Index	Direction Label Sequence	Line Type
S_1	"*up*"	Side line with up gradient
S_2	"*down*"	Side line with down gradient
S_3	"*straight*"	Straight line
S_4	"*up*", "*straight*"	Up corner
S_5	"*straight*", "*up*"	Down corner
S_6	"*down*", "*straight*"	Down corner
S_7	"*straight*", "*down*"	Up corner
S_8	"*up*", "*straight*", "*down*"	Upper half-circle
S_9	"*down*", "*straight*", "*up*"	Down half-circle

Table 4.2: Possible direction labels for each line after *Split*.

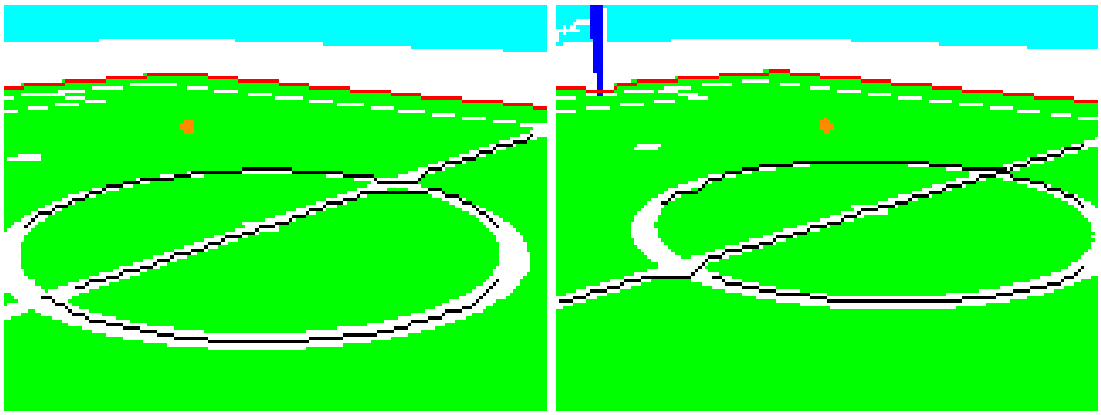

Figure 4.10: Lines formed before (left) and after (right) split and merge.

Figure 4.10 shows how useful split and merge can be. In the left image, line formation wrongly added the side line to the upper-half circle line, but as we can see in the right image, with the use of split and merge, the correct line formation was established.

4.5.4 Line Classification

At this stage we are trying to extract the information we need for each detected line in the image. In the interest of distinguishing a curved line from a straight one, we applied the least-squares fit method, explained in Section 2.3.1. The data provided for

Thresholds	**Conclusion**
If $(a > 0.7)$ && $(a < 5.2)$ && $(b < 0.6)$ && $(b > -0.6)$	Upper Half-Circle
else if $(a < -0.7)$ && $(a > -5.2)$ && $(b < 0.6)$ && $(b > -0.6)$	Down Half-Circle
else if $(a >= 5.2)$ && $(b < 0.6)$ && $(b > -0.6)$	Up corner
else if $(a <= -5.2)$ && $(b < 0.6)$ && $(b > -0.6)$	Down corner
else if $(b > 0.07)$	Side line with down gradient
else if $(b < -0.07)$	Side line with up gradient
else	Straight line

Table 4.3: Thresholds on least-squares fit parameters for line classification.

least-squares fit of a degree-2 polynomial includes the pairs (w_i, h_i) for all black lines i belonging to the line, where w_i is the width coordinate and h_i is the height coordinate of the midpoint of the corresponding black line. The parameters a, b, and c returned by the method are provided to the *ProcessLine* function, which compares them with certain thresholds and ultimately returns the type of the detected line. Parameter a is scaled to 1000 times of its original value to make the comparisons easier. These thresholds were determined after a lot of experimentation and deep consideration in order to make the function as successful as possible. *ProcessLine* can also identify corners that weren't able to be broken in the previous subsection. Table 4.3 presents how the thresholds are used for line classification.

When a line has been identified, a particular pixel of each line type is sent to estimate the direction and distance to the corresponding part of the line, as explained in Section 4.2:

- For side lines with gradient or not, we send the middle pixel of the line.
- For an upper corner, we send the pixel of the line that is highest in the image (the lowest height coordinate), whereas for a down corner we send the pixel that is lowest in the image (the largest height coordinate).
- Because curved lines are only detected in the center circle area, when a curved line is found, we are attempting to position the circle's center, so we can estimate distance and direction from that point. However, the field center circle is projected

as an ellipse in our image, so we cannot estimate distance and direction directly, like the other line types. We choose three points from the detected curved line (first, middle, and last), we estimate distance and direction to each of these three points, we convert these polar coordinates to Cartesian coordinates on the ground, and finally we apply the method described in Section 2.3.2 to fit a circle to these three points on the ground. This method returns the (c_x, c_y) coordinates of the circle's center in regard to our position. Finally, the estimated distance and direction to the center of the circle can be calculated as follows:

$$CircleDistance = \sqrt{{c_x}^2 + {c_y}^2}$$
$$CircleDirection = \arctan 2(c_y, c_x)$$

It is important to mention, that because of the great similarity between corners and curved lines, when a circle section is detected from the *ProcessLine* function, a procedure is called to verify the existence of a curved line. This procedure breaks the line in two equal parts and reapplies the least-squares method on each part using a quadratic polynomial. Depending on the new a parameters, it generates a new estimation about the original line type. If a_1 and a_2 are the parameter values of the quadratic term of the two half lines scaled by a factor of 1000, then a curved line is verified if $|a_1| > 1$ and $|a_2| > 1$. If a curved line cannot be verified, then we have to check if our line is a corner or a side line. So, if $|a_1| < 1$ and $|a_2| < 1$, then our original line is taken as a side line. Finally, if none of the above holds, then the original line is taken as a corner.

Furthermore, a *DetectCornersAndJunctions* function is implemented to decide if two neighboring side lines form a corner or a junction:

- **FindCorner** First, we check that the directions of the two side lines can form a corner. For example, two side lines that are both marked as *up* cannot do so. Then, we find the smallest distance between the two ends of the two lines. This way, four different corner types can emerge:

  1. *Up corner* The distance between the end of $line_1$ and start of $line_2$ is the smallest, the direction of $line_1$ is *up*, and the direction of $line_2$ is *down*.
  2. *Down corner* The distance between the end of $line_1$ and start of $line_2$ is the smallest, the direction of $line_1$ is *down*, and the direction of $line_2$ is *up*.

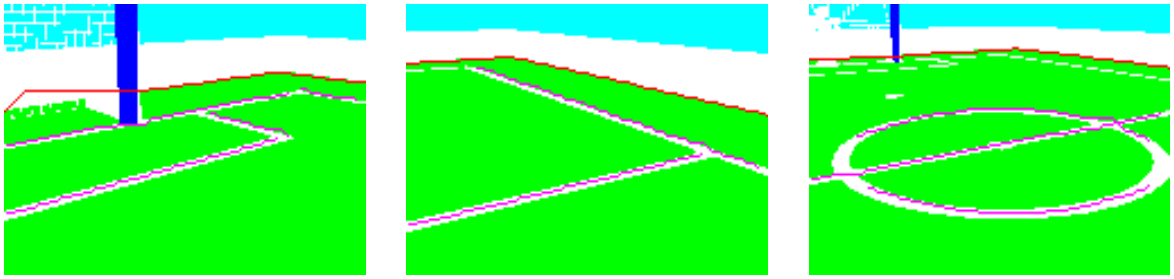

Figure 4.11: Line recognition: right/up corner (left), junction (middle), and circle (right).

3. *Right corner* The smallest distance is between the end of $line_1$ and the end of $line_2$.

4. *Left corner* The smallest distance is between the start of $line_1$ and the start of $line_2$.

Of course, the above are not enough to conclude the existence of a corner. Additionally the two ends of the lines that form the corner must be close enough (no more than 10 pixels in width, and no more than 5 pixels in height). Finally, when we are sure that a corner is detected, we choose the pixel corresponding to the corner, for estimating direction and distance from the corresponding corner landmark on the field.

- **Find Junction** Only two junctions of the field are recognized and used in the localization process; they are spotted on the middle line of the field. Their detection process is simpler compared to the rest of the landmarks and it is based on checking the start and end coordinates of each line. If one of these coordinates is within a certain distance from the other line (no more than 5 pixels in height), then we conclude that a junction is recognized. For estimating distance and direction the pixel at the junction point is used.

Figure 4.11 presents detected lines. The magenta line is the fitted line calculated by the least-squares method.

4.6 Validation

We explained in Section 3.1 the disadvantages of the virtual camera. The small resolution of 160×120 pixels cannot offer precise object recognition and therefore observation errors appear. The deviation in the estimation of distance and direction from a landmark depends on the range and the direction angle of the robot from the specific landmark. We tried to assess the quality of landmark recognition by using a GPS sensor on the robot (not allowed in an official game) to obtain the true coordinates $(GPS_x, GPS_y, GPS_\theta)$ of the exact position and orientation of the robot in the field and plotting the observed landmarks as perceived from that position in the field. The observation error is presented as the discrepancy between the true and the projected position of each landmark. Figures 4.12, 4.13, 4.14, 4.15, 4.16 present five such scenarios of landmark recognition from various field positions. All estimations were taken in the absence of other robots in the field, using a horizontal scan of five image frames. Notice that almost all landmarks within the range of the scan are correctly recognized and the observation error is quite small. We did not quantify observation errors further, because these errors are ultimately included within the deviations in self-localization, which is the target of our work and we thoroughly evaluate.

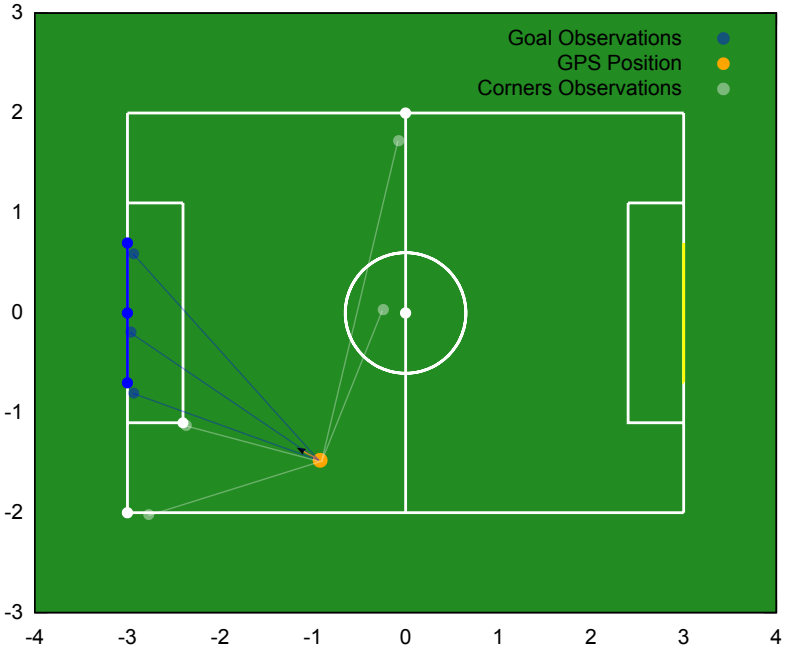

Figure 4.12: Observation errors example 1.

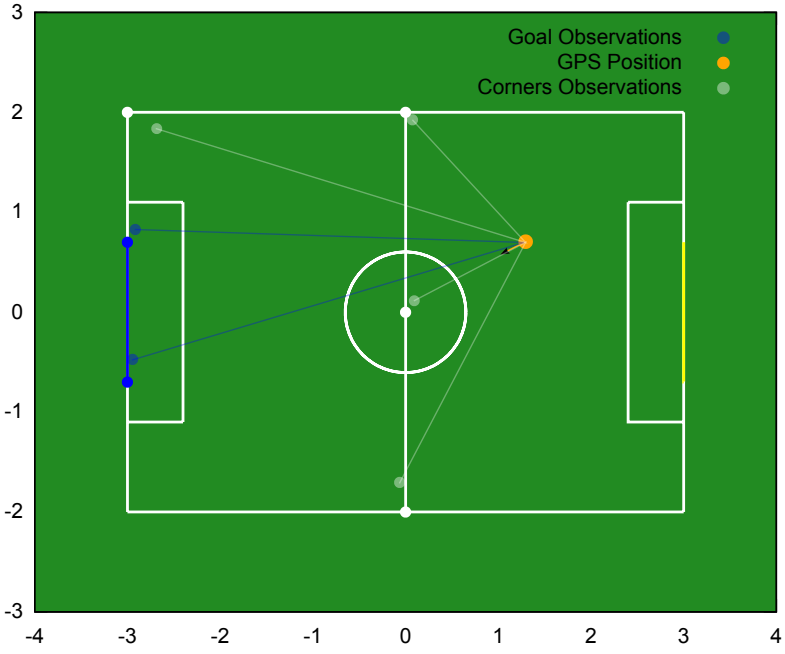

Figure 4.13: Observation errors example 2.

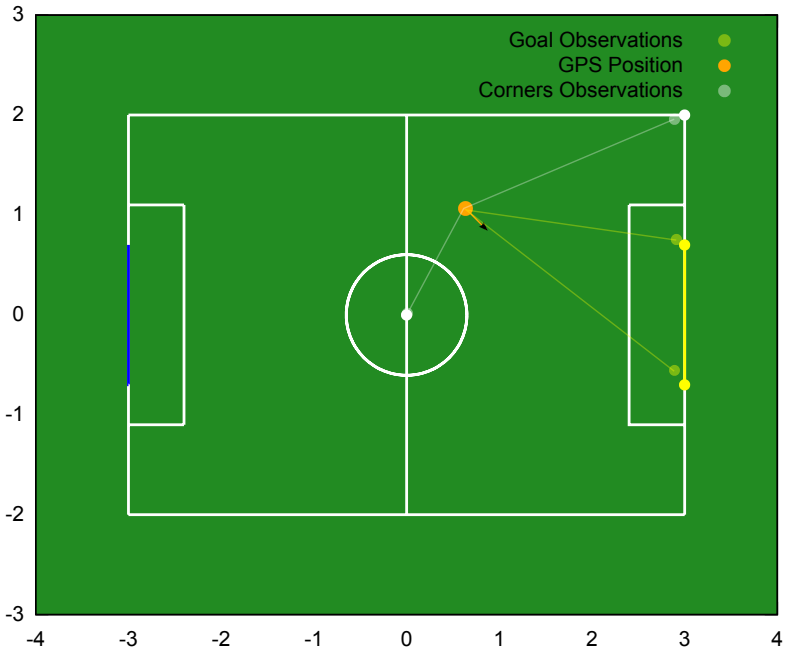

Figure 4.14: Observation errors example 3.

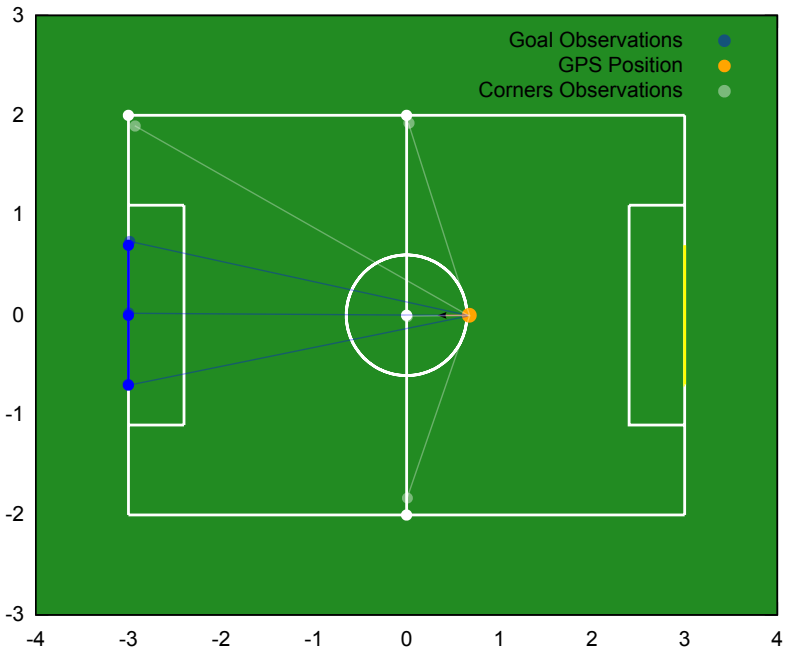

Figure 4.15: Observation errors example 4.

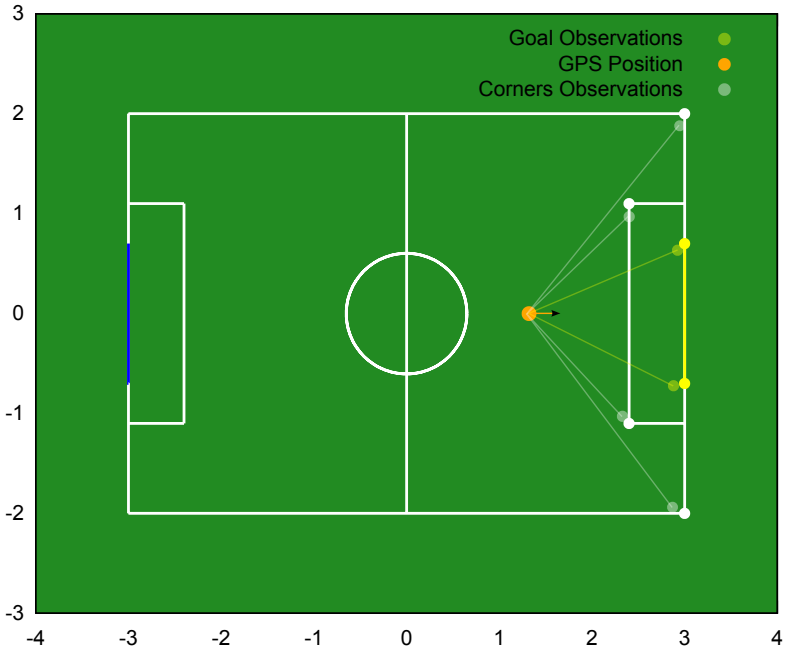

Figure 4.16: Observation errors example 5.

Chapter 5

Self-Localization

In this chapter we present how the principal objective of this thesis is achieved. Up to this chapter we have explained how the image processing works with the aim of creating a dependable landmark recognition module for our agent. This module is useless until it is exploited in order to help our agent become a better football player. In my opinion the most important utilization of vision is a self-localization module. Without it, the player moves in the field and makes decisions exclusively depending on the ball position, having no conviction about his position. Therefore, he is unable to make more complex decisions involving his team. On the contrary, if he was equipped with a self-localization module, he could participate in global decision making for his team's best interest, such as game strategy, team coordination and role assignment among the players. In this thesis however the further exploitation of self-localization is not covered.

The self-localization method we use is constrained-based, meaning that we are trying to pinpoint our agent's location using the constraints imposed by the observations of landmarks. So, for our method to be as efficient as possible, we need to obtain as much information from the field as possible. Given that the camera's horizontal view is rather narrow, we are forced to pan the player's head from left to right in pursuance of scanning a larger percentage of the field. Figure 5.1 presents the increase of the scan angle from 46°, which is the horizontal field of view angle, to 206° providing us with the advantage of recognizing more landmarks. Five image frames with little overlap are taken at each scan; one straight ahead, two to the left, and two to the right of the robot.

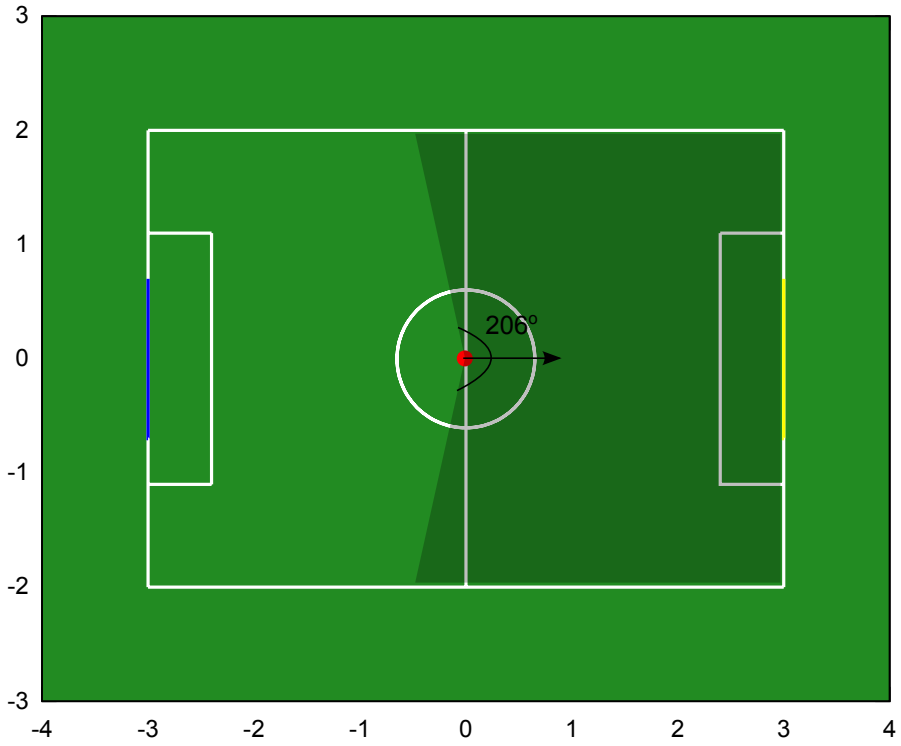

Figure 5.1: Visible scan area.

5.1 Field Coordinates

Our agent's environment is consisted of specific landmarks. Before the problem of self-localization can be addressed, we need to know what the fixed positions of those landmarks are. Each position in the playing field is uniquely defined by a set of coordinates and thus the position of every field object is established by a set of (x, y) coordinates. Figure 5.2 shows these fixed positions. The origin of the axes is located at the field's center, the x axis runs from the blue goal to the yellow goal and its range is $[-3, 3]$, and the y axis runs along the middle line of the field with the blue goal on the left and its range is $[-2, 2]$. To describe the robot's position in the field uniquely, we additionally need an orientation. The zero degrees angle is defined towards the yellow goal, turning left means positive angle, and turning right means negative angle. Figure 5.3 presents the zero orientation angle.

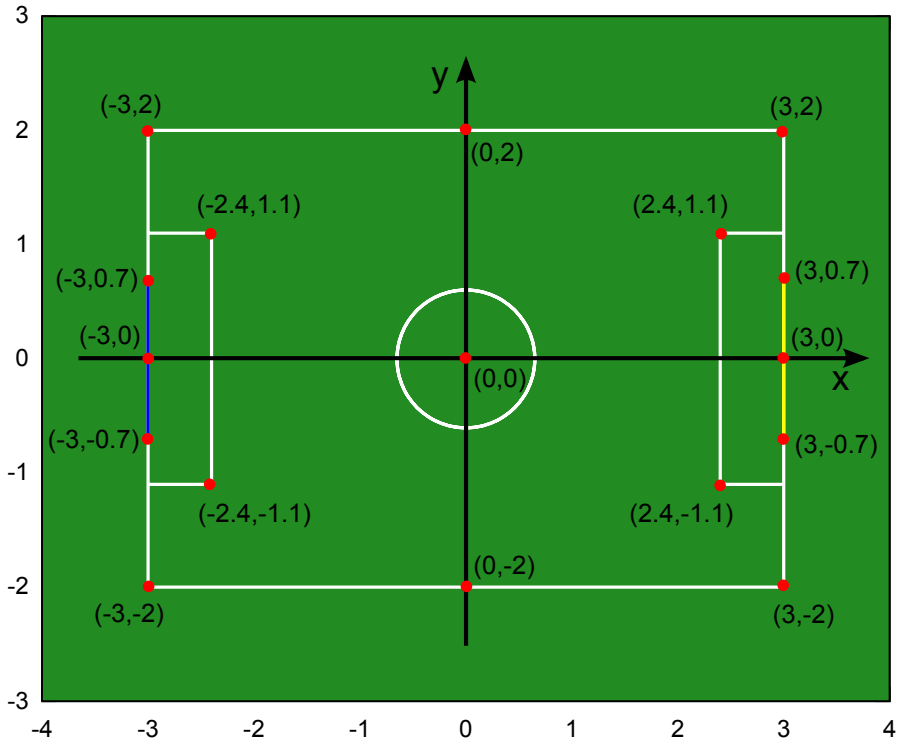

Figure 5.2: Landmarks coordinates on the RobotStadium field.

5.2 Landmark Disambiguation

Some landmarks, such as the goals, the goalposts and the field center, can be positioned on the field directly after recognition, because they are unique and we know in advance what their coordinates are. Corners and junctions however, cannot be treated like the rest of the landmarks. When an up corner is detected, we are unable to know in advance which one of the existing four field corners was recognized due to the fact that they are almost identical to each other. Consequently, a procedure has the responsibility to disambiguate a detected corner or junction. This procedure calculates the angle differences between the current corner and all detected goalposts. The smallest angle difference defines which goal post is closer to our detected corner. Depending on this goalpost and the corner type we can easily infer the unique position of the detected corner. For example, if the smallest angle difference of an up corner is calculated from the blue left post, then this corner is located at the $(-3, -2)$ coordinates of the field. In order to avoid any false detected

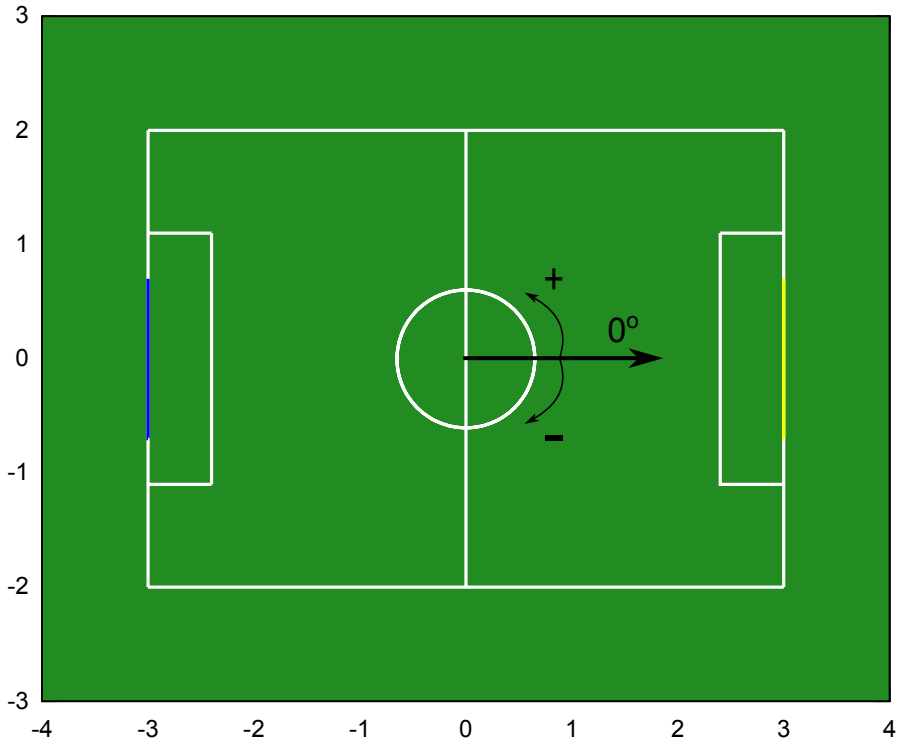

Figure 5.3: Orientation system on the RobotStadium field.

corners, if the smallest calculated angle is larger than a certain threshold (default value 51°), then the corner is rejected. This procedure for landmark disambiguation is also used for the two junctions. Obviously, if no goalpost is visible, then no corner or junction can be positioned. The *normalizeAngle* function in the algorithm takes as input an angle and wraps it within the $[-\pi, \pi]$ range.

5.3 Landmark Observation Constraints

Our constrained-based self-localization method tries to pinpoint the robot's location using the constraints imposed by the landmark observations. These constraints generate several candidate field positions, which are subsequently filtered to keep the one that seems to be best. In the interest of producing the candidate field positions, every time a landmark is recognized, we construct a virtual circle on the field with its center at the landmark's fixed coordinates and with radius equal to the estimated distance between the player and

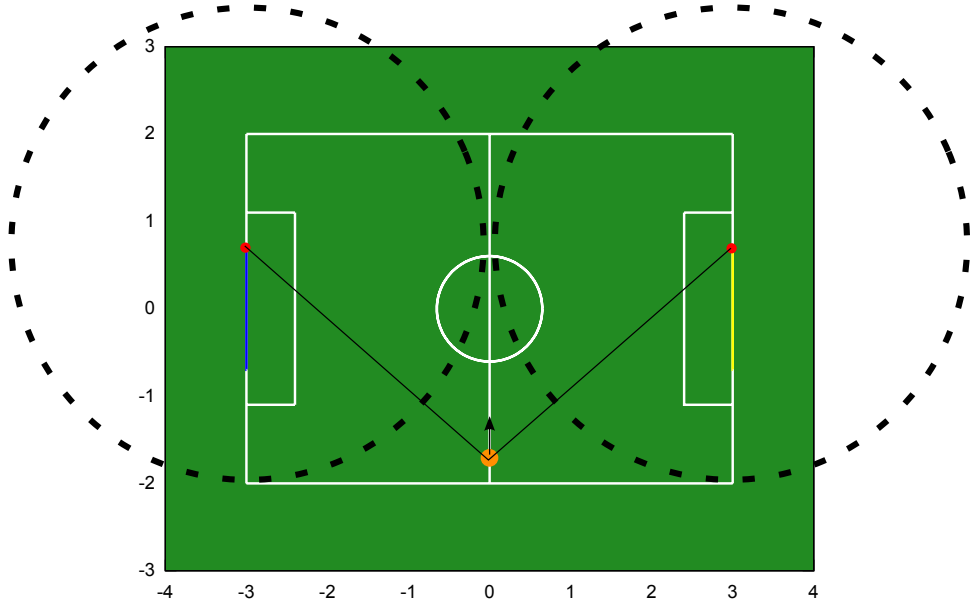

Figure 5.4: Separated circles due to underestimation of distances to landmarks.

the landmark. This circle represents all possible field positions from which this particular landmark can be observed at this particular distance. Thus, when two landmarks are recognized and, therefore, two circles are constructed, the intersection points that emerge between these circles represent two candidate positions for our player. If more than two landmarks are detected, then candidate positions are estimated pairing all landmarks with each other. Apparently when only one landmark is visible, we are unable to extract any candidate positions.

Due to deviations in the landmark recognition module, circles are not perfectly aligned on the field and the candidate positions do not necessarily match the true one. If d is the distance between the two circle centers, and r_0, r_1 are the radius of circle 0 and circle 1 respectively, then five different occasions can emerge:

- **Circles are separated** When $d > r_0+r_1$, the two circles have no intersection points and consequently no candidate positions. It occurs rarely when both landmarks are far and therefore the error in the estimated distances is large. An example is shown in Figure 5.4, where the distance to the two landmarks has been underestimated.

- **Circles are coincident** When $d = 0$ and $r_0 = r_1$, there is an infinite number of intersections and so this occasion is also rejected.

- **Circles are tangent** If $d = r_0 + r_1$ or $d = |r_1 - r_0|$, then there is a tangent point between the two circles. In the first case, the circles are externally tangent, whereas in the second case they are internally tangent. Thus, the desired candidate position is located on one of the circles' circumference. To extract that position, first we have to find the θ angle on the plane indicating the orientation of one landmark with respect to the other. If $C_1(C_{1x}, C_{1y})$ and $C_2(C_{2x}, C_{2y})$ are the center points of the two circles and the landmarks positions respectively, then:

$$\theta = \arctan 2(C_{2y} - C_{1y}, C_{2x} - C_{1x})$$

  Now that if this angle is known we can calculate the $P(P_x, P_y)$ point. If r_1 is the radius of the larger circle, then:

$$P_x = r_1 \cos(\theta) + C_{1x}$$
$$P_y = r_1 \sin(\theta) + C_{1y}$$

  Figure 5.5 presents the above method for internally tangent circles in addition to an example. In the first image the circumference point marked in red is a possible location of our agent.

- **Circles are contained within each other** If $d < |r1 - r0|$, we have no intersection points. However, in this case we can compute the error between the two distance estimations, subtract it from the larger circle radius and make the two circles internally tangent, in which case we can use the solution applied in the previous occasion. We chose this resolution because typically the error is caused by the distance estimation error of the farthest landmark. Assuming that r_0 is the large circle's radius then the error e is computed as $e = r_0 - (d + r_1)$. After subtraction the radius of the larger circle becomes equal to $d + r_1$. Figure 5.6 offers an example of this occasion.

- **Circles intersect** If none of the above holds then two candidate positions can be extracted using the method explained in Section 2.3.3. Figure 5.7 illustrates two instances of this occasion. In the first image one of the candidate positions falls outside the field's border and is directly rejected, whilst in the second image both candidate positions will be kept.

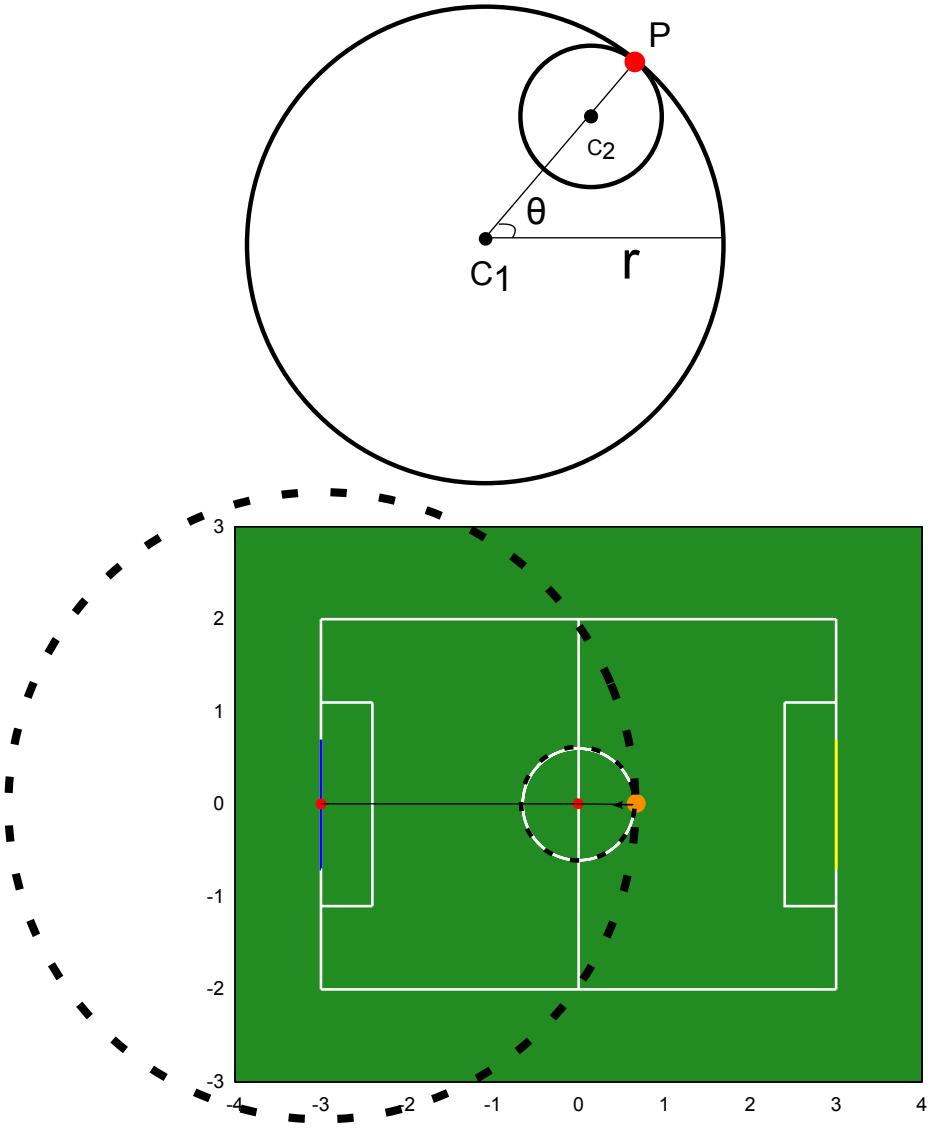

Figure 5.5: Circles internally tangent yielding a single candidate position.

5.4 Candidate Position Filtering

After all candidate positions have been extracted, we need to determine which one is the closest to the real position of the robot. Candidate positions that fall outside the field's borders are rejected during the previous stage. Additionally, we have to find the player's orientation in the field. For this task it is essential to calculate the orientation of each recognized landmark in the orientation system of the field from each candidate position.

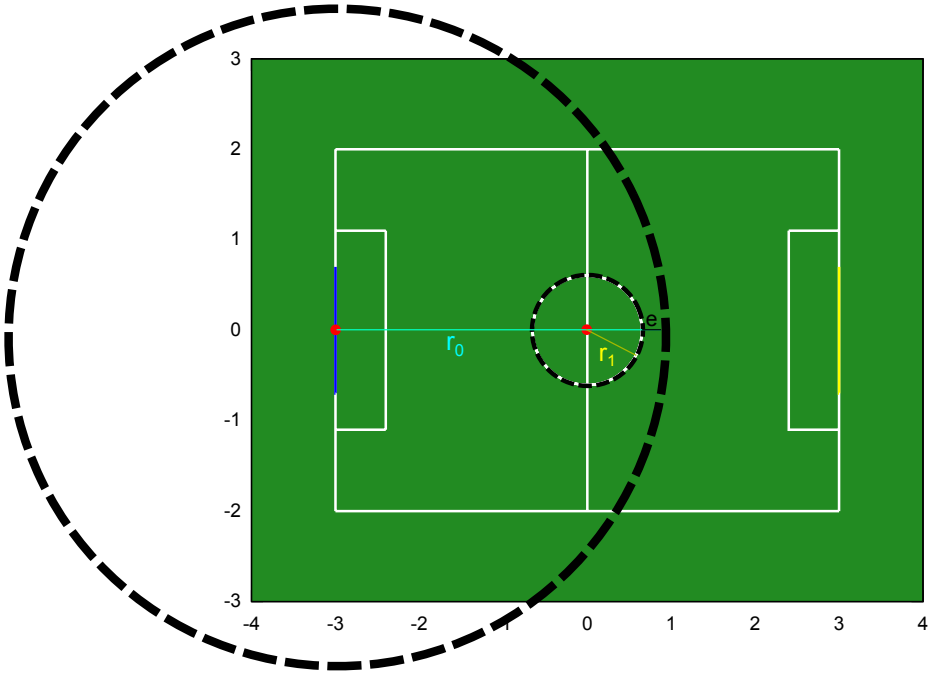

Figure 5.6: Contained circles due to overestimation of distance to landmark.

For each candidate position and for each recognized landmark, we compute the robot's orientation in the field by subtracting the observed direction angle to that landmark from the computed orientation of that landmark in the field. If that candidate position was the true one, all recognized landmarks would yield the same robot orientation in that position. However, since most of them are incorrect, different recognized landmarks would yield completely different robot orientations in the same candidate position. In fact, even the best candidate position, due to observation errors, would not yield identical robot orientations. Therefore, for each candidate position we find and store the maximum difference between all pairs of the resulting robot orientations. The candidate position yielding the least of these maximum differences is chosen as the best candidate. In other words, we are seeking to obtain the candidate position that best matches all observation angles of the recognized landmarks. Finally, the best robot orientation is taken from the best candidate position. Algorithm 4 describes this procedure in detail. Next we will present a series of examples:

1. In the instance of Figure 5.8 our agent was in a place to recognize the blue goal and its posts, the field center, two corners in the edges of the field and the two junctions

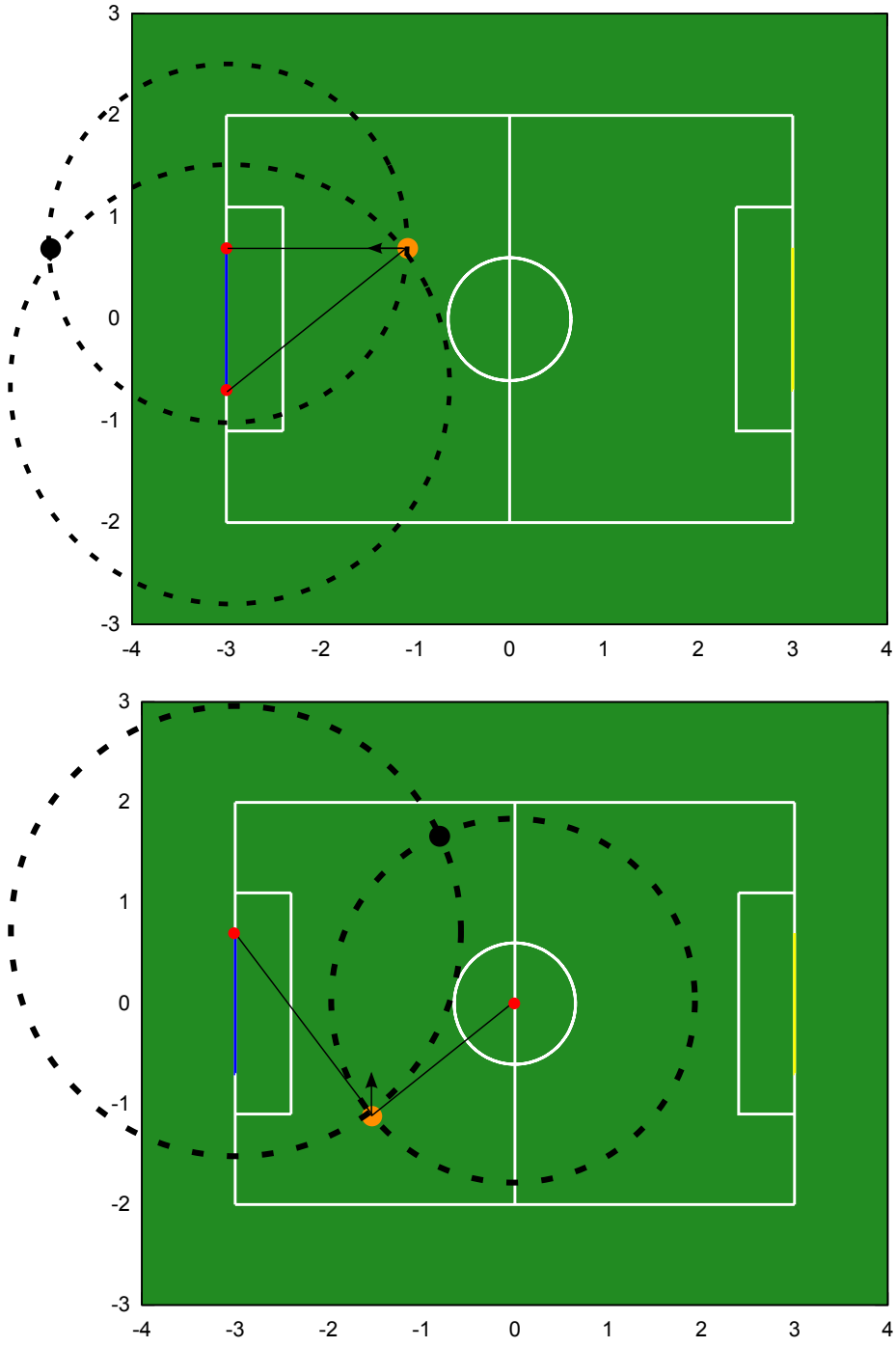

Figure 5.7: Intersecting circles yielding two candidate positions.

due to the widening of the scan area. We notice that the majority of the candidate positions is very close to the real one, and the chosen position is almost identical.

2. In the example of Figure 5.9 we were able to recognize the two goal area corners besides the blue goal and its posts and the two corners on the edge. Similarly as the previous example, most of the candidate positions are gathered in a small area, very close to the real robot's position. However, due to errors in the calculation of direction the chosen position is not the best, but it is still satisfactory.

3. In Figure 5.10 we observe that although most of the field was visible, we were able to recognize and use only five landmarks: blue goal left and right posts, yellow goal left and right posts, and field center, because as mentioned in Chapter 3 our camera's resolution is small, rendering object recognition from far not dependable. Nevertheless our best estimation is almost identical to the robot's real position.

4. In the instance of Figure 5.11 we recognize the yellow goal posts, the two field corners and the center circle. We notice that even though the full field's center was not in our scan area, a small part of it was visible in our image, making it possible for a curved line to be detected.

All the above figures present various localization examples, illustrating the candidate positions, the chosen best position, the robot's real position and the landmarks recognized in the process. During these examples the agent was alone and moving randomly in the field.

5.5 Odometer

A very useful module in self-localization is the odometer. Its function is to update the position and orientation of the agent, every time he makes a move. This way we obtain another source of estimating our position, although it is not as dependable as the localization procedure we described. This is due to the fact that errors accumulate, as the odometer updates its values in consecutive time steps. Figure 5.12 displays the aforementioned effect. Notice the rapid increase of error in the odometer values in both occasions. This happens because the odometer takes into consideration the effect that every motion has on the agent's movement which cannot be perfectly measured. For

Algorithm 4 Find Best Position and Orientation

Input: $Positions, RecognizedLandmarks$
Output: Best Robot Position and Orientation
for all $p \in Positions$ **do**
  for all $l \in RecognizedLandmarks$ **do**
    $l.fieldDirectionAngle = \arctan 2(l.y - p.y, l.x - p.x)$
    $\theta = normalizeAngle(l.fieldDirectionAngle + l.observedDirectionAngle)$
    $Orientations.add(\theta)$
  end for
  $p.orientation = Orientations(0)$
  $p.maxDiff = 0$
  for all $(\theta_1, \theta_2) \in Orientations \times Orientations$ **do**
    $diff = \left|normalizeAngle(\theta_1 - \theta_2)\right|$
    $p.maxDiff = \max\left\{p.maxDiff, diff\right\}$
  end for
end for
$best = \underset{p \in Positions}{\arg\min} \left\{p.maxDiff\right\}$
return $(best.x, best.y, best.orientation)$

example, a motion that supposedly turns our agent 40° right, it doesn't actually do so with accuracy on the field; the actual turn angle may be anywhere between 35° and 45°. Consequently, we avoid to use the odometer too many times without a localization procedure based on landmarks intervening. When a localization outcome is accepted, the odometer resets its values accordingly. The odometer is only used in the following occasions:

- When the player recognizes less than two landmarks and so a candidate position cannot be extracted.
- When all candidate positions are rejected before being able to choose the best candidate position.
- When the best candidate position is more than half meter away from the odometer position. However, if this occurs for a second consecutive time, then the odometer values will be reset at the new position. This is done to prevent receiving false

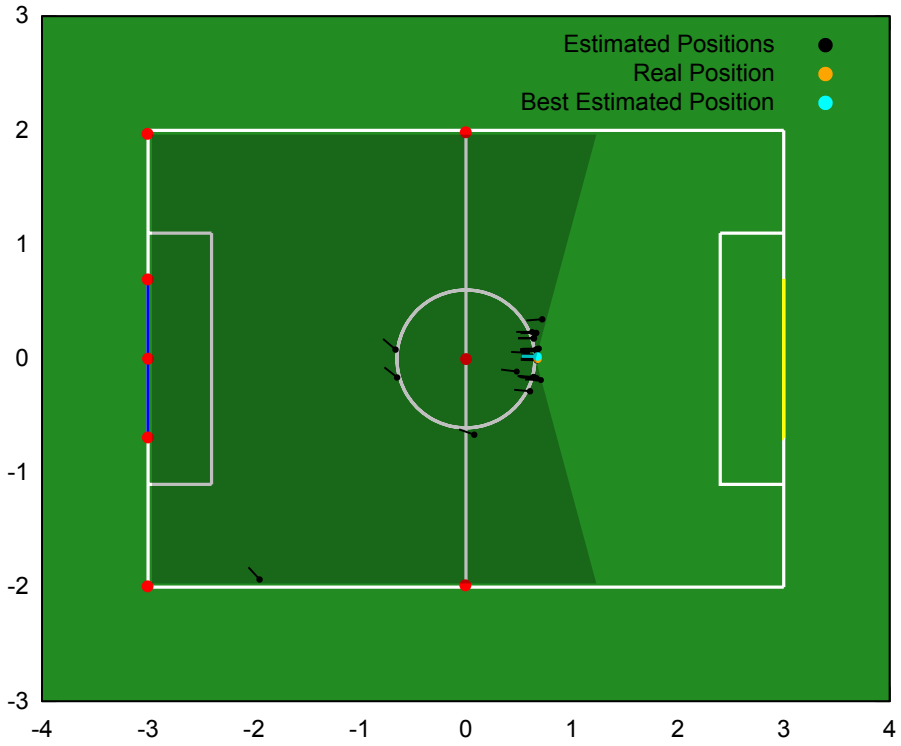

Figure 5.8: Constraint-based localization example 1.

positions from the odometer, when the player falls on the ground or is being moved by an external factor (i.e the supervisor).

The odometer updates the agent's orientation by adding the turn angle of the last performed turn motion to the already existing orientation angle. If this angle is not in the $[-\pi, +\pi]$ range, then it is normalized. The new position is updated depending on the last performed move motion and the current orientation. If it is a forward motion then:

$$Odometer_x = Odometer_x + \cos(\theta) \times d$$
$$Odometer_y = Odometer_y + \sin(\theta) \times d$$

and if we are talking about a side step motion then the x,y estimations are reversed:

$$Odometer_x = Odometer_x + \sin(\theta) \times d$$
$$Odometer_y = Odometer_y + \cos(\theta) \times d$$

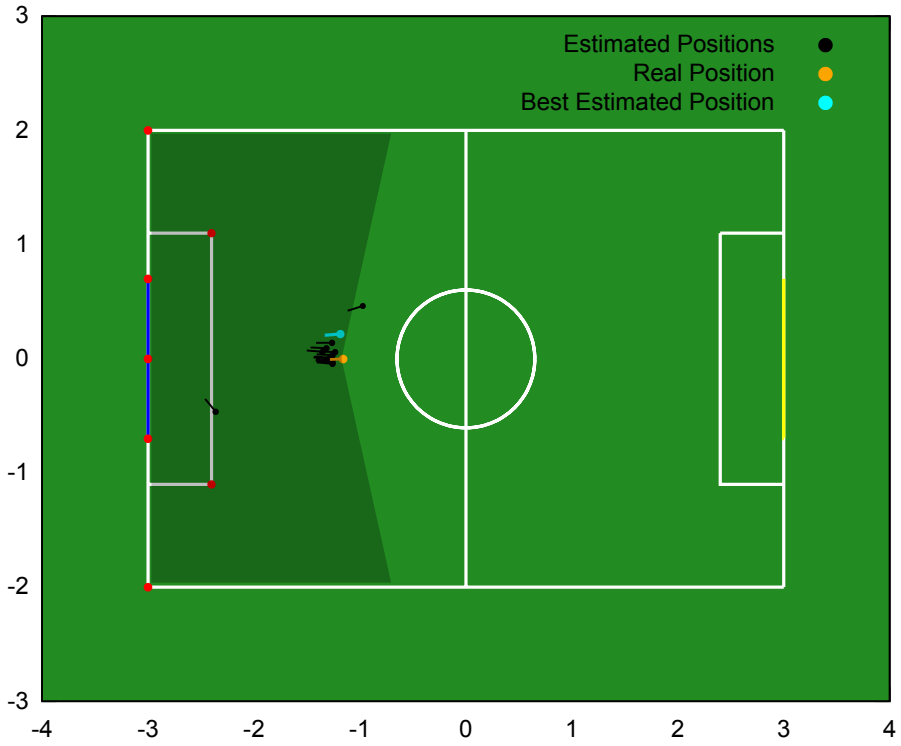

Figure 5.9: Constraint-based localization example 2.

where $Odometer_x$ and $Odometer_y$ are the current odometer coordinates, θ is the current orientation angle value and d is the distance traveled by the motion. The initial odometer values are set to the predetermined positions depending on whether our agent is in the blue or red team and whether it has the kick-off or not. At each kick-off the odometer values are reset.

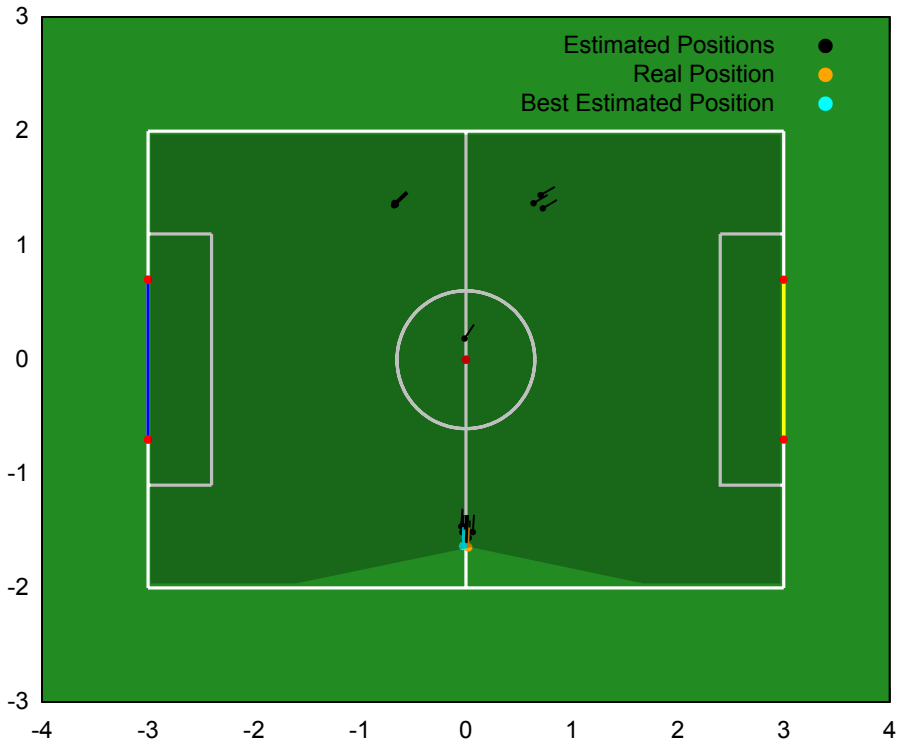

Figure 5.10: Constraint-based localization example 3.

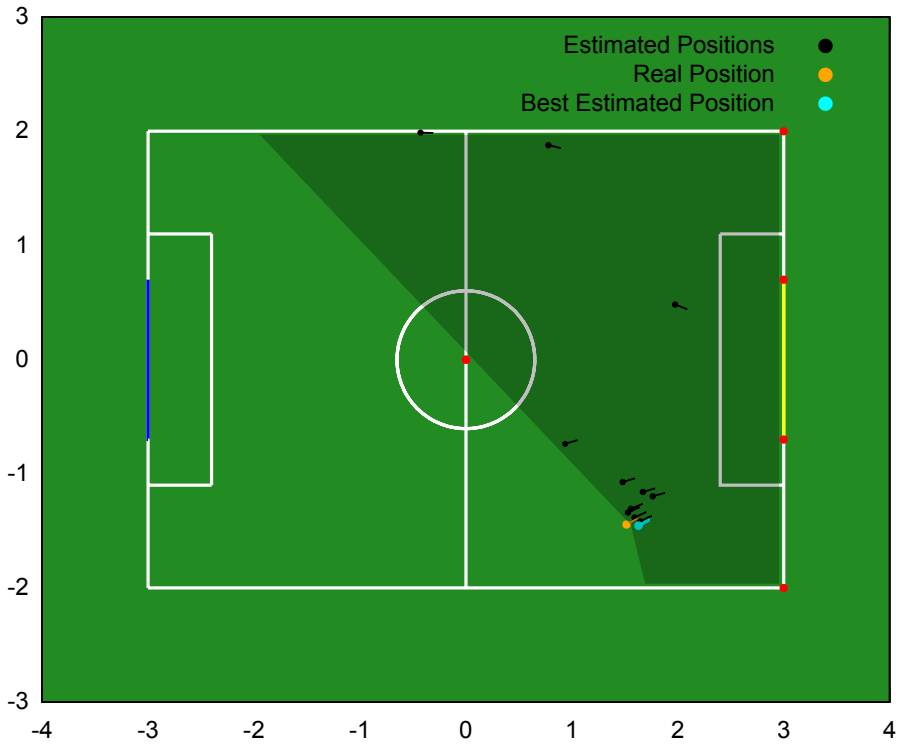

Figure 5.11: Constraint-based localization example 4.

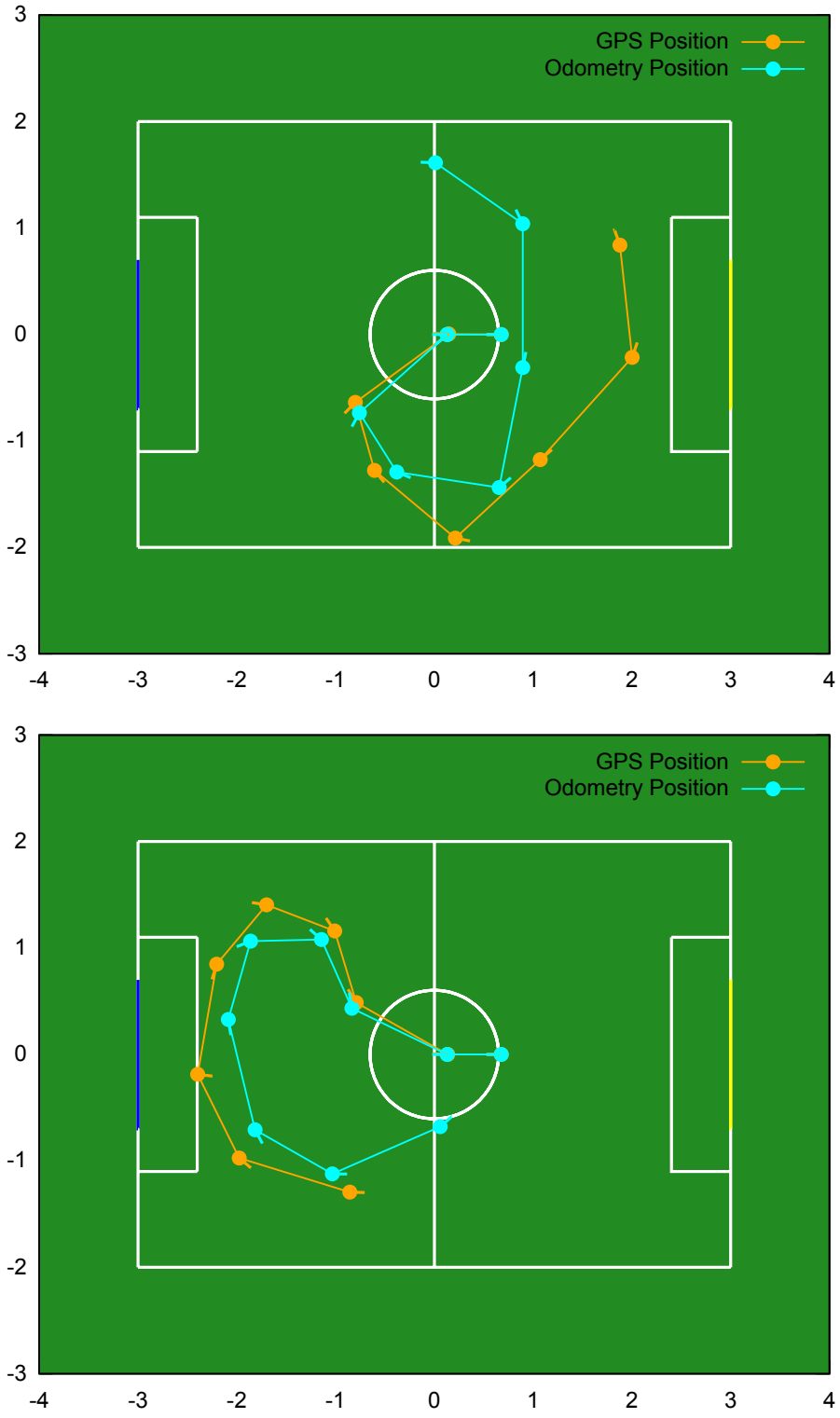

Figure 5.12: Odometer values in comparison with the agent's real position.

Chapter 6

Results

In this chapter we will demonstrate a variety of tests in order to present our self-localization module's performance. The tests are built around three scenarios: one agent moving around in the field, one agent against two opponents, and a full game of four players on each side. The reason for this variety is to display the different levels of difficulty an agent faces depending on the number of other robots in the field. In all three occasions the simulation is let to run independently without any interference and at its conclusion we collect the data we need, which are the agent's real position via GPS and our estimated one, for comparison.

6.1 Football Player Behavior

A football player behavior is required for our agent to participate in the aforementioned scenarios. For the purposes of this thesis we employed a behavior that was formerly developed during the course of "Autonomous Agents" in the Technical University of Crete, in collaboration with my colleague Konstantinos Hatzipetrou [13]. It uses most of the sensors and actuators that were presented in Section 2.2.3 in order to accomplish the following:

- Detect when the player has fallen to the ground and also with what side so that it chooses the appropriate motion to stand up.
- Detect any obstacles in front of it, recognize if the obstacle is an opponent, a teammate, or a goalpost and try to avoid it.

- Recognize and track ball during the game, with the aim of approaching it and attempting to shoot.
- Recognize and focus on opponent goal when ready to shoot.
- When a shoot motion is performed, detect if the ball was actually kicked. If not, try to gain a better position for shooting.

In general this agent implements basic field player behavior. Slight modifications allowed us to add the landmark recognition along with the self-localization module. In the enhanced behavior, we simply enabled our player to kick the ball towards the opponent goal without trying to visually locate it first. This is made possible, because it is straightforward to compute the direction angle to the opponent goal, when our player knows his position and orientation in the field. There are several opportunities to utilize a dependable localization module, such as team formations, roles, strategies, etc., however these fall beyond the scope of this thesis.

6.2 Scenario 1: One Agent in an Empty Field

An agent is let to freely choose a random motion and move in the field. The GPS and estimated positions are extracted just before our agent makes a move. So, after each move, the agent performs a scan and localizes himself. This is a very simple scenario, representing the best case, because we have no other players in the field to obstruct our agent's vision or cause misplacement. This scenario serves as the baseline of evaluating our work. Figures 6.1, 6.2, 6.3, 6.4, 6.6 feature several examples of this scenario. We observe that some self-location estimations are precise, while others are only close to the real position. There is no case, where the estimated self-location is totally wrong. Notice that the worst estimations are made when the agent is farther from any landmarks and, therefore it, is more possible for deviations to occur.

6.3 Scenario 2: One Agent Against Two Opponents

In this scenario our agent is playing against two field players of the opponent team. The total number of robots on the field is five including our player and, of course, the

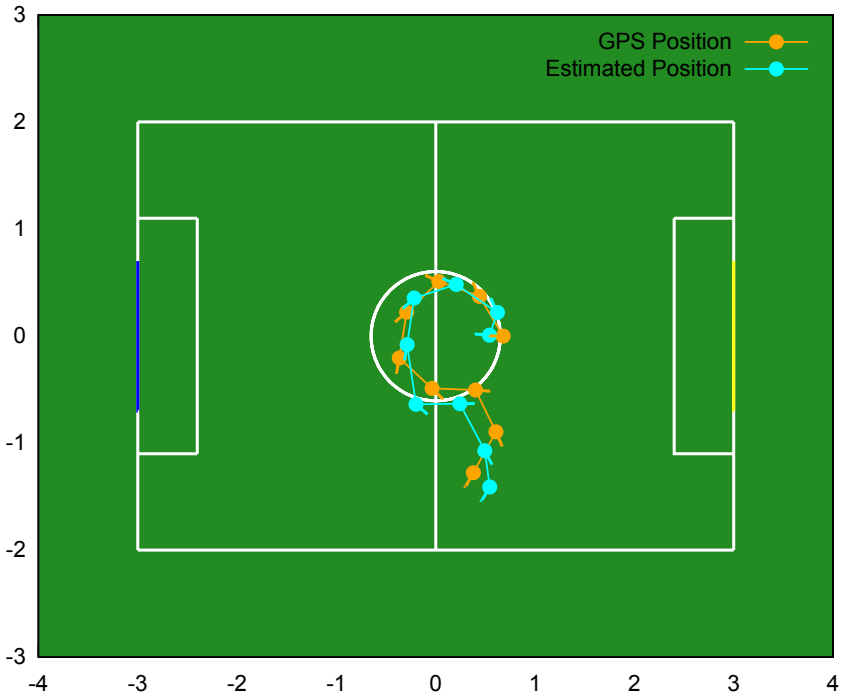

Figure 6.1: Scenario 1: localization of a single player in an empty field (example 1).

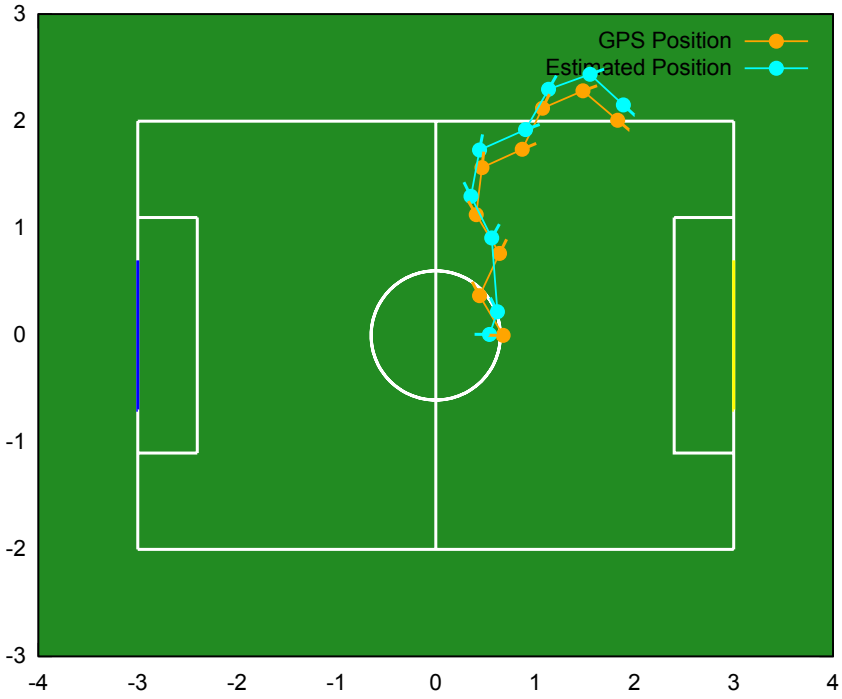

Figure 6.2: Scenario 1: localization of a single player in an empty field (example 2).

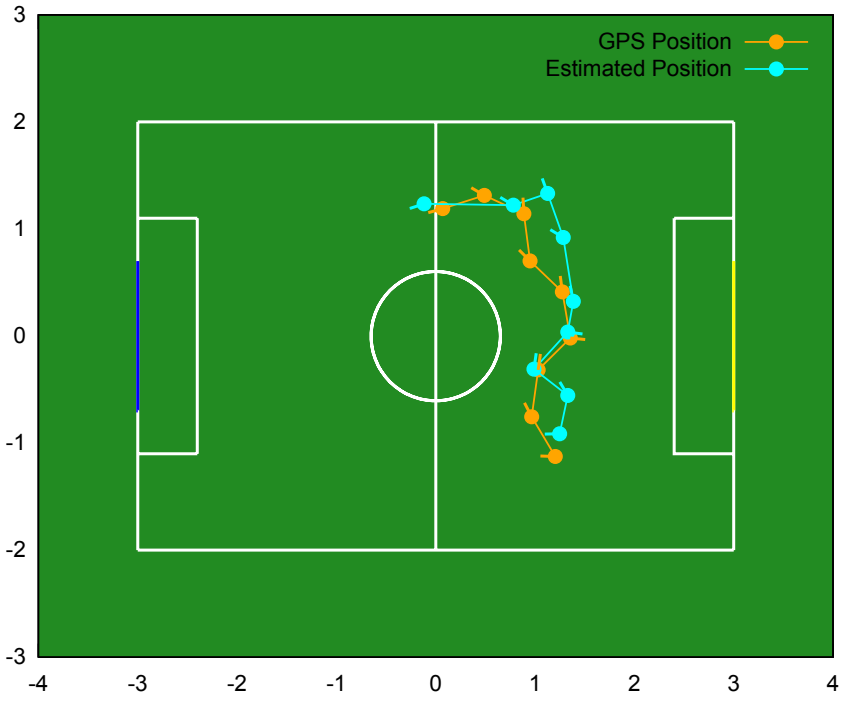

Figure 6.3: Scenario 1: localization of a single player in an empty field (example 3).

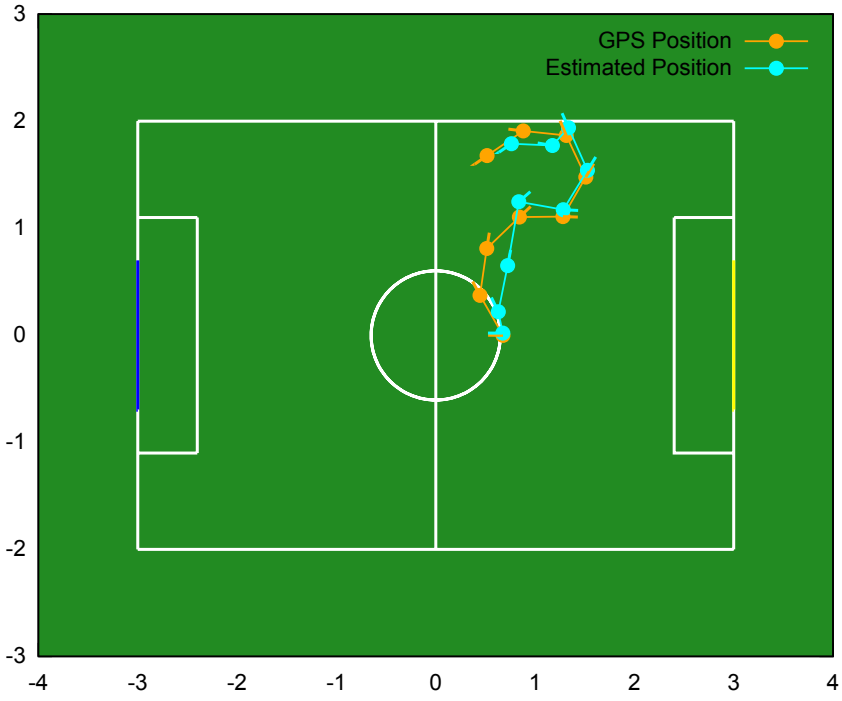

Figure 6.4: Scenario 1: localization of a single player in an empty field (example 4).

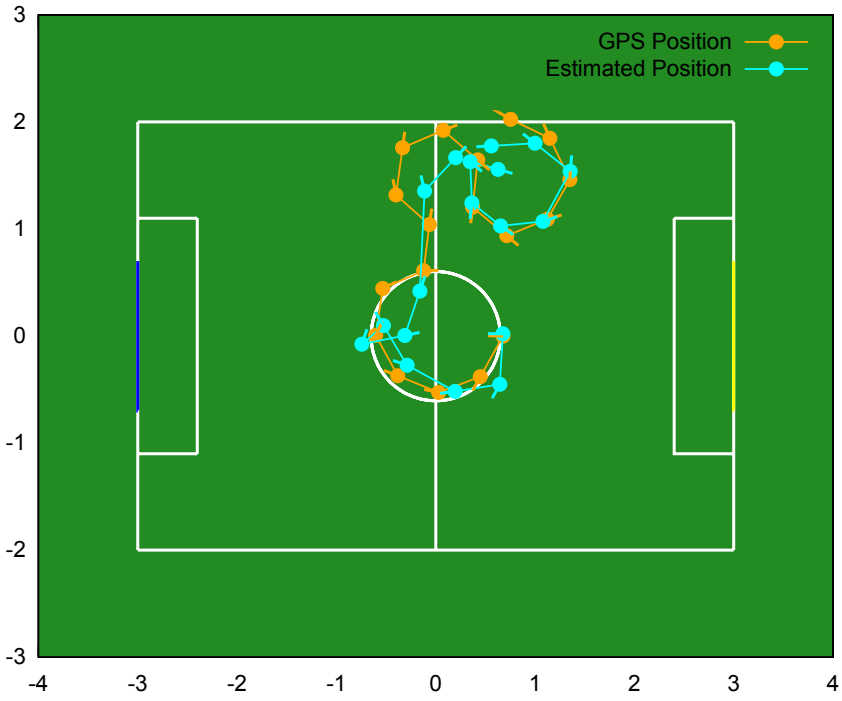

Figure 6.5: Scenario 1: localization of a single player in an empty field (example 4).

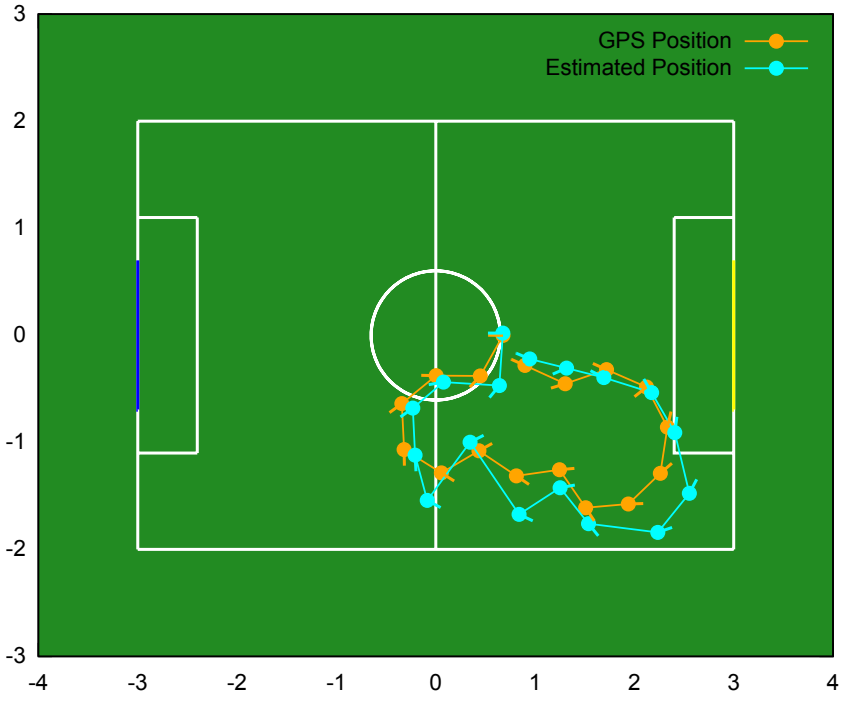

Figure 6.6: Scenario 1: localization of a single player in an empty field (example 5).

two goalkeepers. The purpose is to observe how efficient our self-localization method is in a semi-crowded environment. Figures 6.7, 6.8, 6.9, 6.10 offer four examples. It is obvious that our estimations are not as good as those in the previous scenario, but still dependable. When our agent falls on the ground the orientation arrow is not presented. In the second example (Figure 6.8) we observe two totally wrong estimations. These occur when the odometer calculations significantly deviate from the reality due to obstructions in the field. This is similar to the occurrence of robot kidnapping, a well-known problem in robot localization. However, as seen in the same example, the estimation is quickly fixed in subsequent steps.

6.4 Scenario 3: Full Game

Here, we use all available players (three field players and one goalkeeper on each side) to demonstrate a full game between two teams in Robotstadium. Our agent has to overcome several difficulties in order to self-localize, such as low visibility of field landmarks, large deviation of odometer, and high error rates. The opponent team was chosen deliberately not to use their sensors to detect obstacles in their path resulting in multiple collisions between robots in the game. This was done in order to simulate even worse conditions than a Standard Platform League game to test our self-localization method. As a consequence, our estimations are worse compared to the other two scenarios, but, given the circumstances, are still sufficiently accurate and can deal with the kidnapping problem. Figures 6.11, 6.12, 6.13, 6.14 present the estimated and the true traces of all three field players of the team during a full game.

6.5 Evaluation

In order to compare the three aforementioned scenarios, we have obtained statistics from multiple executions; these statistics include the average errors in position and orientation estimations out of 50 samples in each case, as well as the 95% confidence intervals. Tables 6.1 and 6.2 show these values and Figure 6.15 displays these results graphically. Our experimental results demonstrate that our approach leads to accurate self-localization with the average error from the true location in position and orientation ranging from $12cm/4^\circ$ in the empty $4m \times 6m$ SPL field to about $42cm/23^\circ$ in the worst case of a

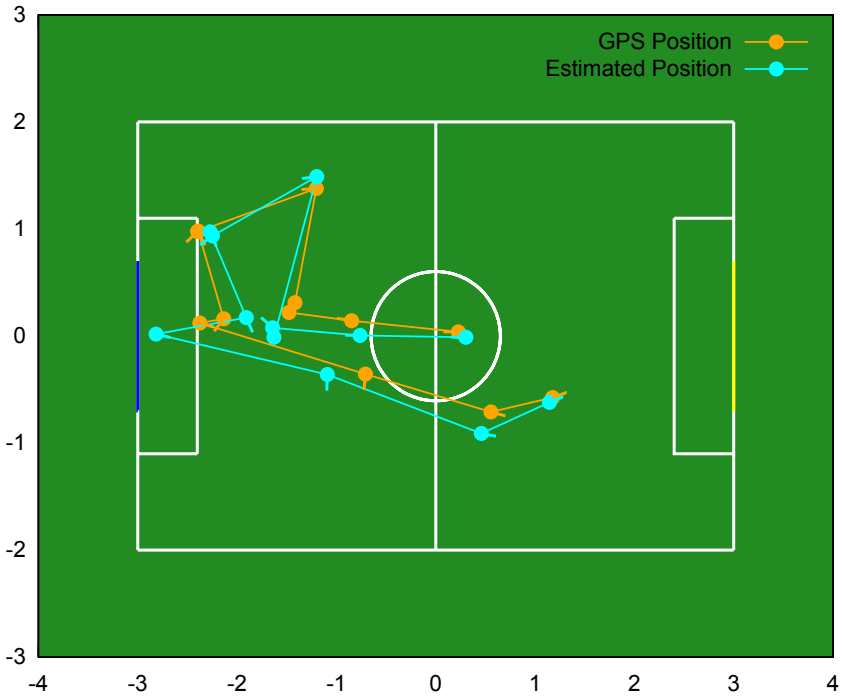

Figure 6.7: Scenario 2: localization of one agent against two opponents (example 1).

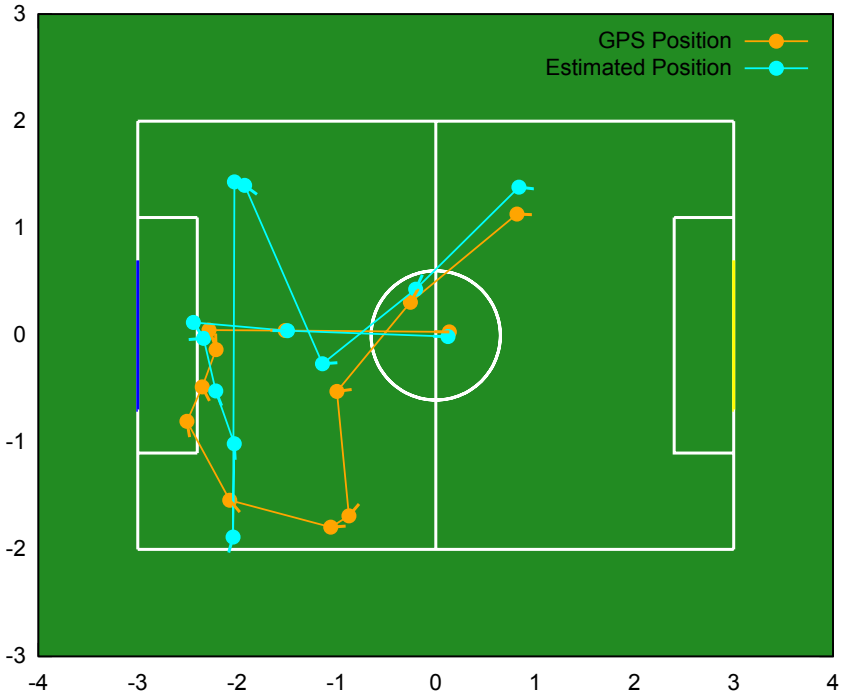

Figure 6.8: Scenario 2: localization of one agent against two opponents (example 2).

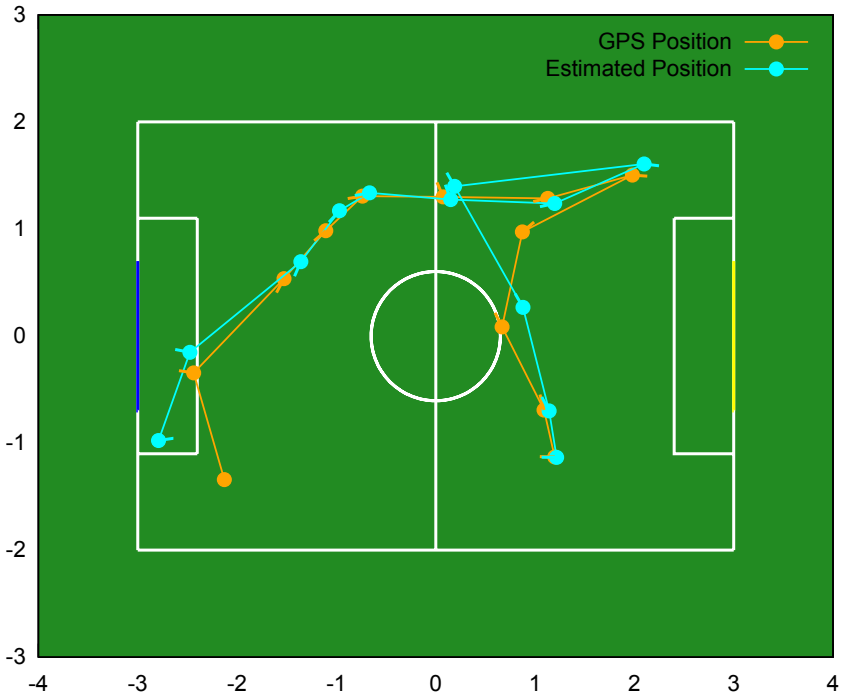

Figure 6.9: Scenario 2: localization of one agent against two opponents (example 3).

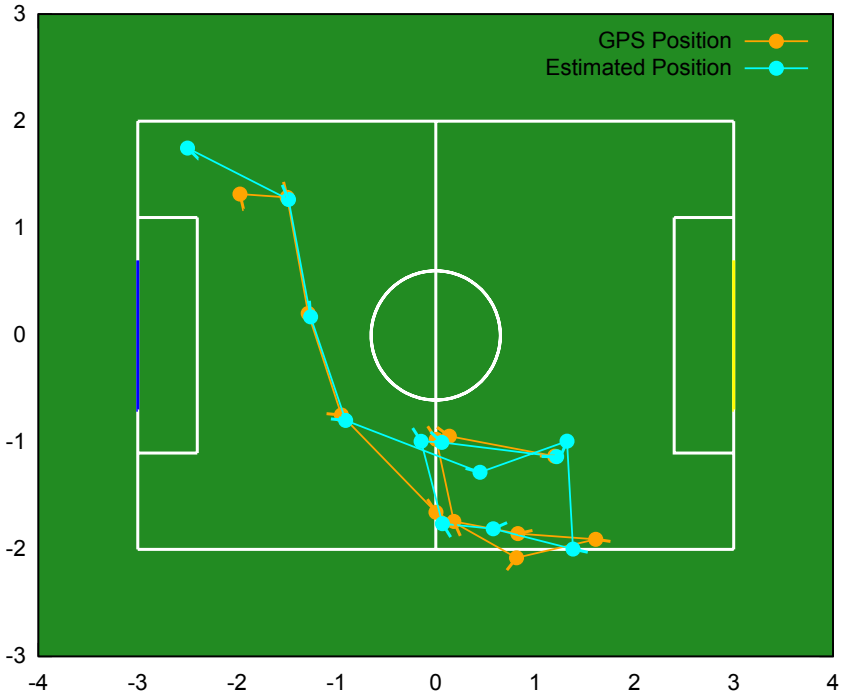

Figure 6.10: Scenario 2: localization of one agent against two opponents (example 4).

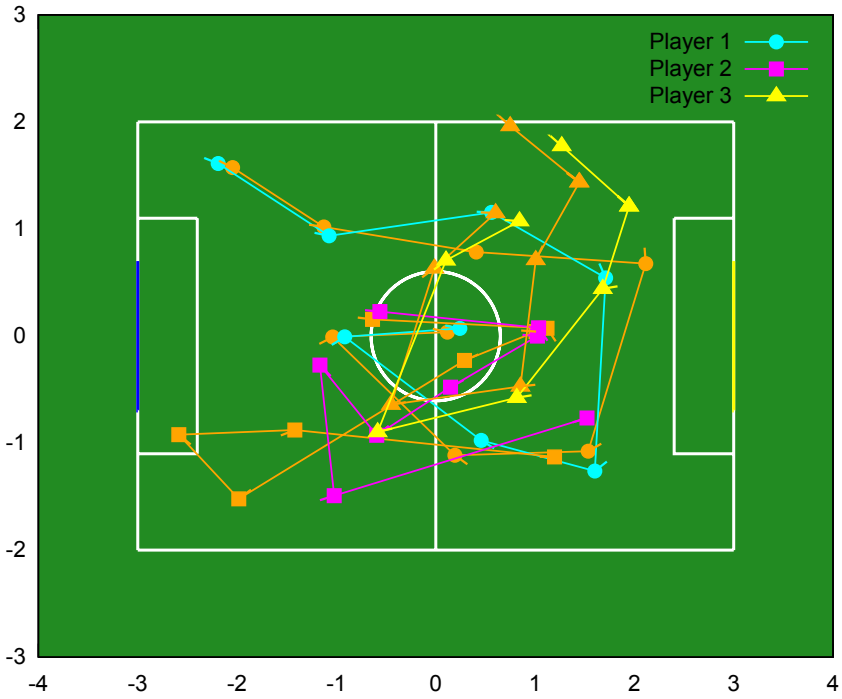

Figure 6.11: Scenario 3: localization of all field players in a full game (example 1).

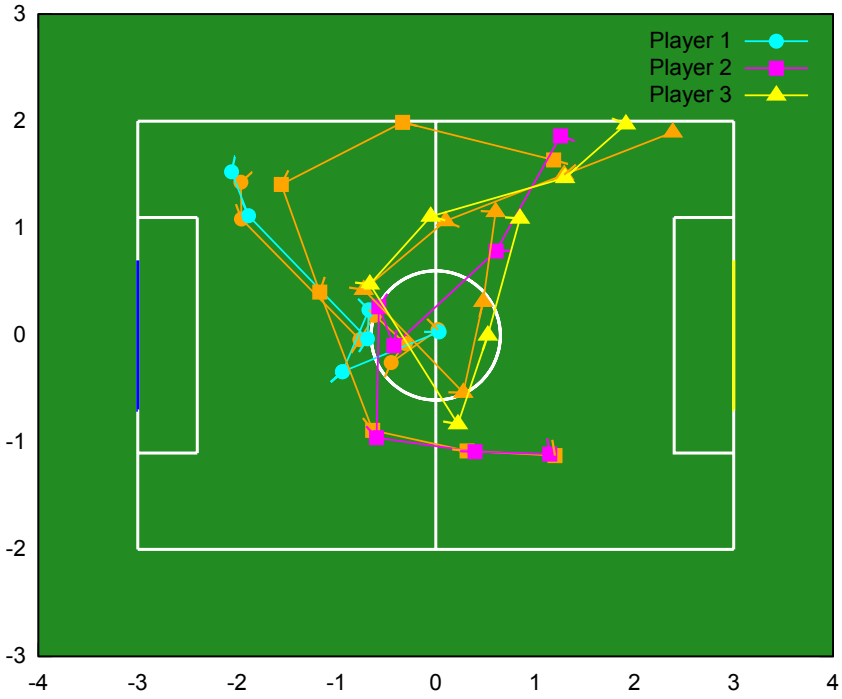

Figure 6.12: Scenario 3: localization of all field players in a full game (example 2).

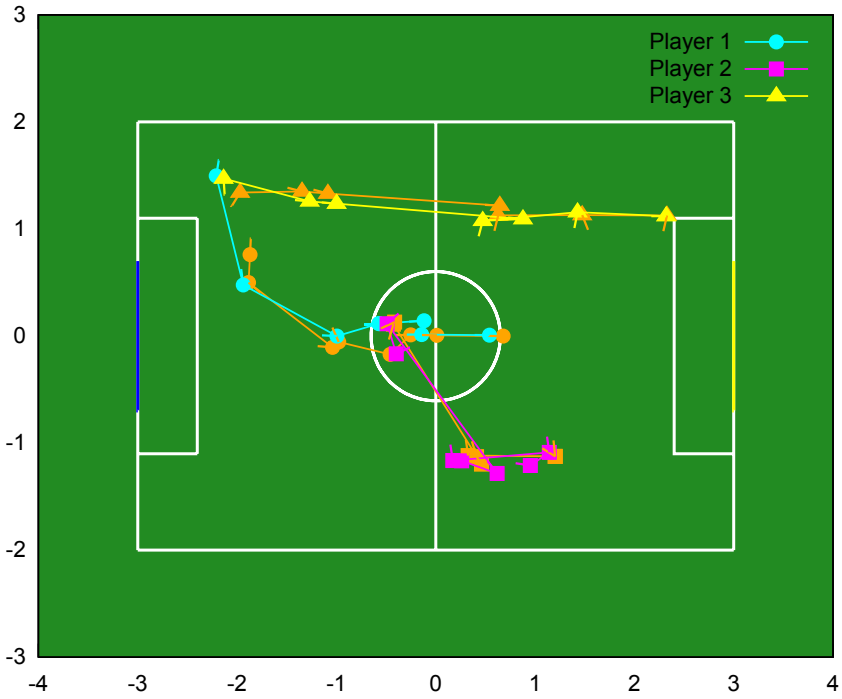

Figure 6.13: Scenario 3: localization of all field players in a full game (example 3).

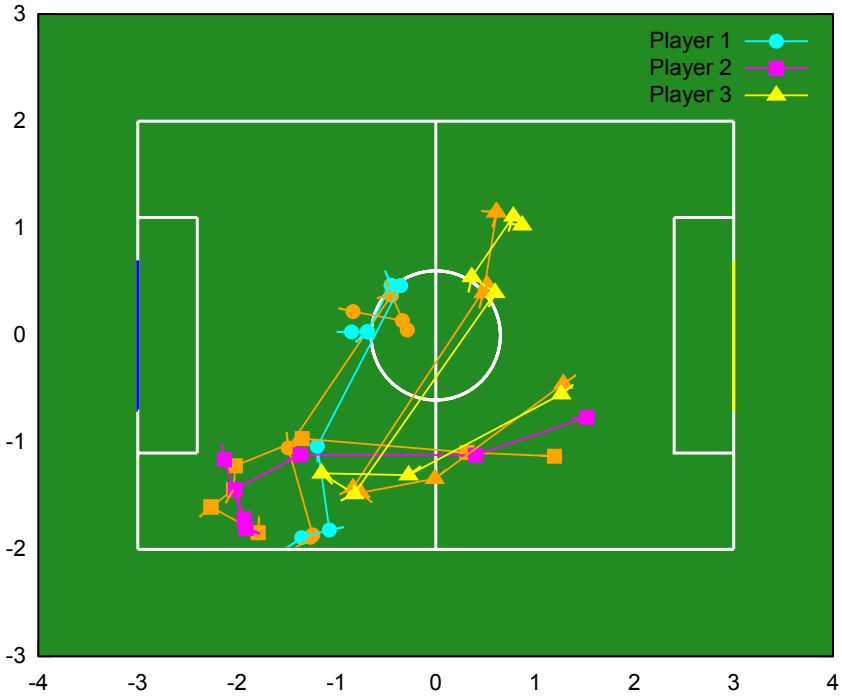

Figure 6.14: Scenario 3: localization of all field players in a full game (example 4).

Position (x, y)	**Mean Error Value** (cm)	**95% C.I.**
Scenario 1	12.39	2.83
Scenario 2	27.83	9.62
Scenario 3	43.09	12.69

Table 6.1: Position error values for each scenario.

Orientation (θ)	**Mean Error Value (degrees)**	**95% C.I.**
Scenario 1	3.53	2.86
Scenario 2	16.32	4.53
Scenario 3	22.91	11.20

Table 6.2: Orientation error values for each scenario.

full game. We observe that the number of players in a game significantly affects our self-location estimations. The mean error values are quite low, when the agent is moving alone in the field, but become high in the full game scenario, where visual obstructions and misplacement due to pushing by other players are highly likely.

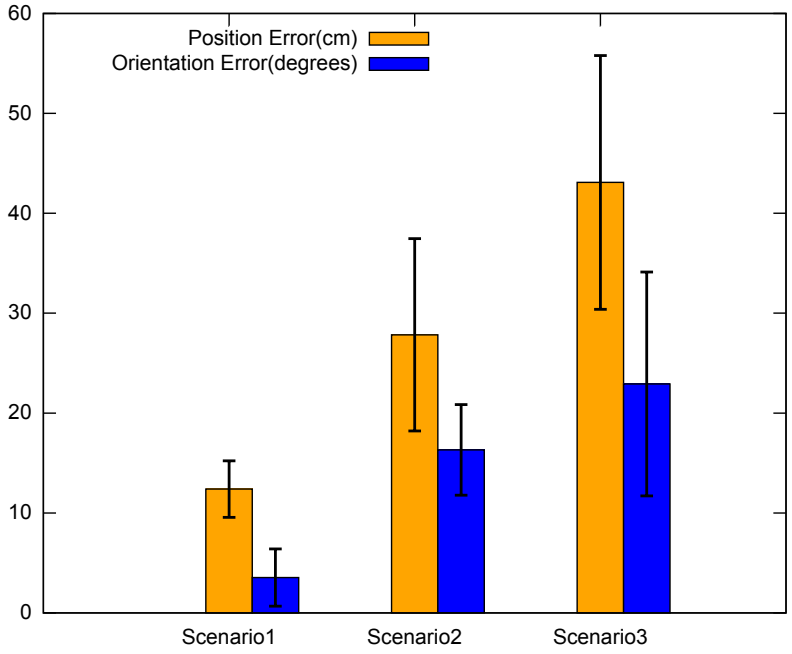

Figure 6.15: Mean position and orientation error values for each scenario.

Chapter 7

Related Work

Given that there is insufficient information regarding how other RobotStadium participants developed their teams, we expanded our search to any form of robotic football simulation that embodies our field of research; landmark recognition and self-localization. Some related work by other teams is presented shortly.

7.1 B-Human Standard Platform League Team

B-Human [14] is currently one of the most successful teams participating in the Standard Platform League (SPL). It was founded in 2006 by the University of Bremen and the German Research Center for Artificial Intelligence (AI). B-Human has won the SPL championship in RoboCup 2009, RoboCup 2010, and RoboCup 2011. In their 2011 code release they explain how their landmark detection module works and how self-localization is implemented.

First, they run scan lines starting from the field border to the bottom of the image to detect any non-green areas and they use them to create segments. Then, they create regions by iterating over all segments and connecting the current segment to an existing region or to a new one. Figure 7.1 displays this procedure. Two segments of the same color touching each other need to fulfill certain criteria to be united to a region: they cannot be larger than a maximum region size, the length ratio between them must not exceed a threshold, there must not be a change of direction, and, finally, if they're already connected to a region, they cannot be united. Next, they classify the created regions depending on whether they are parts of a line or the ball. Here, we will describe only

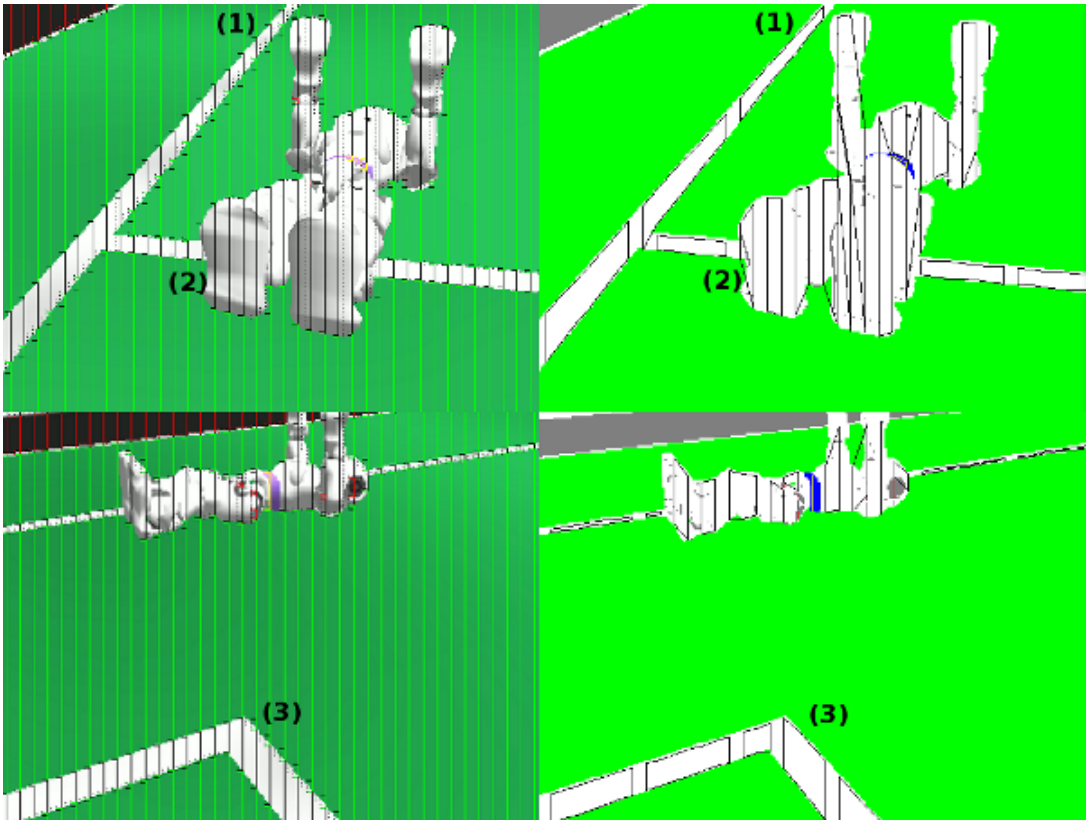

Figure 7.1: B-Human 2011 SPL team: usage of scan lines and region building.

the conditions needed for a region to be accepted as a part of a line: the region has to be of a certain size containing a certain number of segments, a certain amount of green pixels must be detected above and below or right and left of the region depending on its orientation, and finally the axis of orientation must be determinable. All regions that are accepted as parts of a line are stored as `LineSpots`. To built the field lines they use the information stored in `LineSpots` and in addition they try to determine the existence of a circle. When the clustering of straight line segments is done, all remaining line segments are taken into account for the circle detection. For each pair of segments with a distance smaller than a threshold, the intersection of the perpendicular from the middle of the segments is calculated. If the distance of this intersection is close to the real circle radius then a spot is created at that position. Then, the same procedure that was used for line clustering is also used to find a cluster for the circle. Finally, all resulting lines are intersected. Depending on where the start and end points of the two lines are, the

intersection type is decided.

To detect goalposts, they scan the projection of the horizon in the image for blue or yellow segments to detect points of interest and attempt to find the foot and head of a possible post. Next, they detect the borders of the post and therefore its width in order to examine if it can be accepted. At the end, they transform the foot and head into field coordinates, and if the distance between these two measurements doesn't exceed a certain threshold the foot coordinates are accepted.

For self-localization, B-Human uses a particle filter based on the Monte Carlo method. This gives them the ability to obtain accurate results and to deal with the kidnapped robot problem, albeit with a more complicated probabilistic method, compared to ours.

7.2 Kouretes 2008 RobotStadium Team

In 2008, Kouretes [15] entered the RobotStadium Competition and managed to qualify for the final stages of the tournament. The field player agent of the team supported basic movements, goal and ball detection, and a behavior module. Self-localization was attempted, but was never included in the actual players. It used a particle filter based on the Monte Carlo method. Due to the fact that only the goal posts were recognized, the update of the particle weights was done in accordance to the goal distance and direction. The estimated self-position was the mean value of all particles whose distance was less than a given threshold from the particle with the highest weight. Because of the limited field landmark recognition, the self-localization suffered from large deviations, especially when goals were not in the visible area of the agent. Figure 7.2 offers an example of the particle estimation when the agent is in the middle of field, looking directly to the goal.

7.3 Kouretes 2013 3D Simulation League Team

Kouretes plan to enter the 3D Simulation League [16] with a recently-developed framework, implementing player behavior and team strategy. The 3D Simulation League is similar to RobotStadium, given that they are both simulations and the football field along with the players are graphically represented. However, the two competitions intend to focus on different problems. In the 3D Simulation League there is no need for

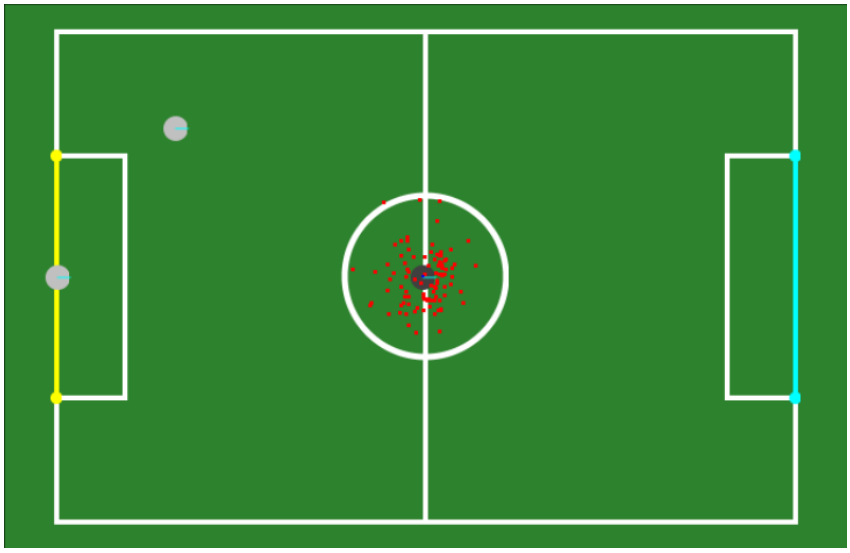

Figure 7.2: Kouretes 2008 RobotStadium team: article filter estimations.

landmark recognition, because the game server is responsible for providing all necessary information of the visible field landmarks to the players.

The self-localization method of this team is very similar to our implementation. For every two visible landmarks, two circles are formed with radius equal to the observed distance and each one of them is centered at the fixed coordinates of these landmarks. These two circles intersect at two points which represent two candidate self locations. Because all landmarks are situated along the border of the field, one of the two candidate locations will always be outside the field limits. When more than two landmarks are visible this procedure iterates between all pairs of landmarks. The final estimated location is computed as the average of the outcomes of all pairs. Figure 7.3 shows an example.

7.4 Dutch Nao Team

Dutch Nao team [17] consists of Artificial Intelligence (AI) Bachelor's and Master's students, supported by a senior staff member. They debuted at the Standard Platform League (SPL) competition at RoboCup German Open 2010. We will examine their self-localization method which uses a dynamic tree [18].

Dynamic tree localization splits the field recursively into blocks. Each node of the tree represents a specific area or region of the field. Each block holds a probability that a robot

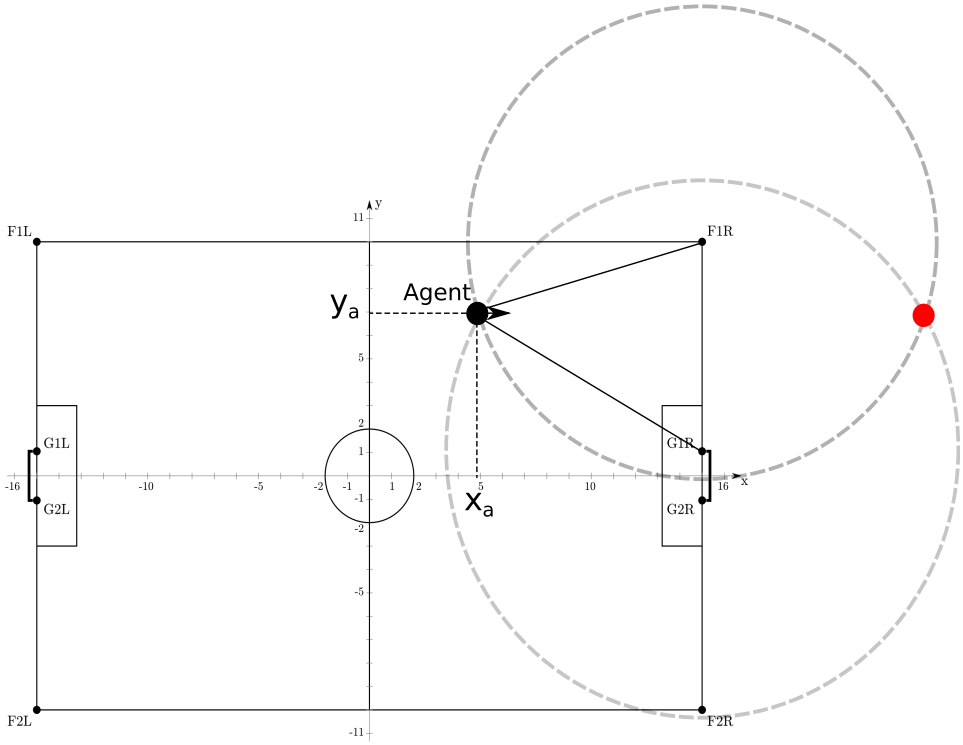

Figure 7.3: Kouretes 2013 3D Simulation team: self-Localization with two landmarks.

is in this block. Besides a probability, a block contains a list with the distance (a range) and the angle (also a range) towards each possible observable feature. As features, the goals and line crossings are used. All observed features are propagated through all nodes in the current belief (the tree) and the probability of a block is updated depending on a feature's attributes. Once all observations are propagated through the tree, sets of rules determine if a block needs to be expanded or collapsed. Collapsing a block results in the removal of all the (grand)children which entails that the accuracy of the area described by the block (and its children) decreases. Expanding works the other way around. When the children of a node are created, a list with all minimum and maximum distances towards all features in the environment is created. Having such a list makes it easy to check if a feature is observable from a certain block. When the expanding process is over, the estimated position is the node with the higher probability. Dynamic tree localization has lower computational complexity compared to other probabilistic approaches, is robust against noisy data, and can handle kidnapping.

Chapter 8

Conclusion

In the course of this thesis, we realized that landmark recognition and self-localization in robotic football are two problems that can be under improvement indefinitely. The fact that both take place in a dynamically, partially observable and noisy environment renders them quite demanding with a high possibility of deviations in estimations. Our self-localization module is proven to be inexpensive in terms of computational resources, in comparison to other methods, because it avoids probabilistic calculations and updates, which are time-consuming. This was crucial because the robot's resources are very limited.

8.1 Future Work

The current landmark recognition and self-localization modules are sufficient at the moment to accomplish our goal in this thesis. However, in the future this work can become the basis of a very competitive RobotStadium team or it can be even modified and applied to real Nao robot teams. Whatever the case may be, there are numerous improvements that can be made. In this section we propose some of them.

8.1.1 Game Strategy

A self-localization method is little used, when it is not exploited to help team play. So, a future installment would be a game strategy component that would embody a coordination protocol among players, a role and responsibility assignment, and a better

team behavior. Our location estimations would navigate our agents to assume defensive or offensive formations and thus gain an advantage towards an opponent team.

8.1.2 Ball and Player Localization

In order to have a global view of the world's state, the coordinates of other players and the ball can be calculated and placed into a general field map, providing us the advantage of planning optimal paths towards the ball, while avoiding obstacles.

8.1.3 Dynamic Movement

Currently, our agent's movements are controlled by predefined motions. This can present problems during a game, due to the fact that the motions are not carried out perfectly in the field. Additionally, there is no way we can detect if the motion was executed at all, when another agent obstructs our player, rendering the odometer useless. To confront this problem we have to create a dynamic locomotion module, which controls the robot's servos in real time, making decisions, and feeding us with information depending on the agent's perception. This way we can be informed accurately of all robot's movements and upgrade our odometer's use.

8.1.4 Application to RoboCup SPL

Our approach seems to be working at a very acceptable level within the simulated environment. An important future direction can be the transfer of our implementation from the RobotStadium platform to the real Nao robot platform and the RoboCup Standard Platform League, in the search of a more efficient and less time-consuming self-localization module for our SPL team "Kouretes".

References

[1] RoboCup: Official site: http://www.robocup.org. 5

[2] Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawa, E., Matsubara, H.: Robocup: A challenge problem for AI. AI Magazine **18**(1) (1997) 73–85 5

[3] Kouretes: Official site: http://www.kouretes.gr. 7

[4] Robostadium: Official site: http://www.robotstadium.org. 8

[5] Marc, P.: Nao programming for the RobotStadium online contest (2009) Only available online: https://unball.googlecode.com/svn/humanoide/Walknao/Nao_Programming_for_the_Robotstadium_On-line_Contest.pdf. 9

[6] Babas, K.: TUC United @ RobotStadium (2011) Semester Project in the Autonomous Agents Course. Only available online: http://www.intelligence.tuc.gr/~robots/ARCHIVE/2011w/projects/Babas/index.html. 9

[7] Cyberbotics: Official site: http://www.cyberbotics.com. 9

[8] Robotics, A.: Official site: http://www.aldebaran-robotics.com. 13

[9] RoboCup SPL Technical Committee: Standard Platform League rule book (2011) Only available online: www.tzi.de/spl/pub/Website/Downloads/Rules2011.pdf. 13

[10] Wolberg, J.: Data Analysis Using the Method of Least Squares: Extracting the Most Information from Experiments. Springer (2005) 16

[11] Casey, J.: "The Circle." - Ch. 3 in A treatise on the analytical geometry of the point, line, circle, and conic sections, containing an account of its most recent extensions, with numerous examples. University of Michigan Library (2001) 17

[12] Weisstein, E.W.: Circle-circle intersection Only available online: http://mathworld.wolfram.com/Circle-CircleIntersection.html. 18

[13] Georgakis, G., Chatzipetrou, K.: Team creation for the RobotStadium competition (2011) Semester Project in the Autonomous Agents Course. Only available online: http://www.intelligence.tuc.gr/~robots/ARCHIVE/2011w/projects/GeorgakisChatzipetrou/. 69

[14] Röfer, T., Laue, T., Müller, J., Fabisch, A., Feldpausch, F., Gillmann, K., Graf, C., de Haas, T.J., Härtl, A., Humann, A., Honsel, D., Kastner, P., Kastner, T., Könemann, C., Markowsky, B., Riemann, O.J.L., Wenk, F.: B-Human team report and code release 2011 (2011) Only available online: www.b-human.de/downloads/bhuman11_coderelease.pdf. 81

[15] Chatzilaris, E.: Robotstadium 2008 participation (2008) Semester Project in the Autonomous Agents Course. Only available online: http://www.intelligence.tuc.gr/~robots/ARCHIVE/2008/projects/Chatzilaris/. 83

[16] Methenitis, G.: Player behavior and team strategy for the RoboCup 3D simulation league. Diploma thesis, Technical University of Crete, Greece (2012) 83

[17] Verschoor, C., ten Velthuis, D., Wiggers, A., Cabot, M., Keune, A., Nugteren, S., van Egmond, H., van der Molen, H., Rozeboom, R., Becht, I., de Jonge, M., Pronk, R., Kooijman, C., Visser, A.: Dutch Nao Team description for RoboCup 2012 (2012) Only available online: http://www.dutchnaoteam.nl/wp-content/uploads/2012/05/Dutchnaoteam_2012_TeamDescriptionPaper.pdf. 84

[18] van der Molen, H.: Self-localization in the RoboCup soccer standard platform league with the use of a dynamic tree. Diploma thesis, University of Amsterdam (2011) Only available online: http://staff.science.uva.nl/~bredeweg/pdf/BSc/20102011/VanDerMolen.pdf. 84