Developing a 2v2 NAO Humanoid Robot Soccer Team

Ruud Adriaans, Renzo Poddighe, Christoph Sinhart, Sjoerd van Steenkiste, Christopher Wittlinger

Department of Knowledge Engineering, bachelor year 3, group 1

January 24, 2013

Abstract

This paper is on a project done by third year bachelor students from the Department of Knowledge Engineering at Maastricht University. It covers the basic aspects of robot soccer using NAO robots. These basic aspects consist of physical movement, but the main focus of this paper is on image processing. Various algorithms and techniques concerning this field are discussed, and evaluated. When put into practice, the conclusion is that noise, computational complexity and physical constraints of the NAO make playing a real soccer game a very difficult goal to achieve.

1 Introduction

This paper is on researching the game of soccer for the NAO humanoid robots. Although the basic aspects of soccer might appear obvious for a human player, they are really challenging when performing them with a robot. This is because the robot does not know anything about the environment and is not able to interpret anything by itself.

In order to enable the NAOs to play soccer, several aspects have to be considered. Physical challenges have to be addressed: the NAOs have to be able to walk up to a ball, and kick it into the goal or to a teammate. However, before the NAOs can walk up to the ball, image processing has to be incorporated first. Because the NAOs should not only be able to detect and locate the ball, they also have to recognize both goals on the soccer field. It is also important that they have a sense of where they are positioned in the field, in relation to the other objects. These objects are teammates, opponents, the goals, and the lines on the field.

Using this information, they should also be able to use (basic) strategies, involving team play, decision making, and scoring. Another challenge is to let the robot perform several tasks simultaneously, for instance moving forward, and scanning the field for the ball or goal at the same time.

1.1 The RoboCup

The RoboCup is the platform for soccer games with all kinds of robots. Its goal is to have a team of fully autonomous humanoid robot soccer players win a soccer game, with the official rules of the FIFA, against the winner of the most recent World Cup, by 2050. The league that NAO robots compete in is called the Standard Platform League. This league was formerly known as the Four-Legged League, using the popular Sony Aibo dog-like robots. Official teams consist of two to four NAO robots.

The official rules of the RoboCup SPL are listed on their official website[10]. A quick overview is also given here. The field, shown in Figure 1, is six meters long and four meters wide. The field is a green carpet with thick white lines on it. The goals on either side of the field are colored in such a way that robots can distinguish between the two different goals. One goal is sky blue and one goal is yellow. A game consists of two halves of ten minutes, and during half time, the teams switch sides.

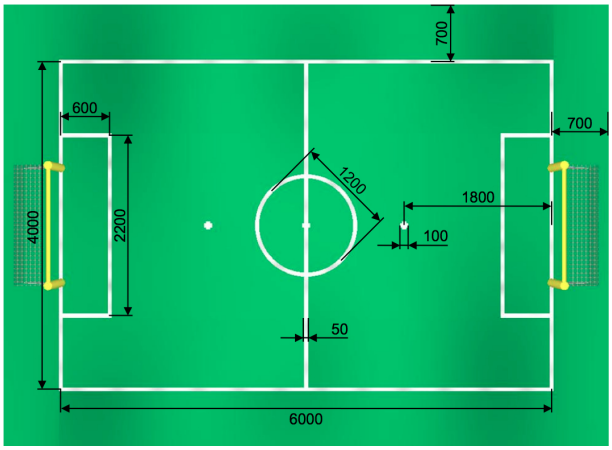

Figure 1: The dimensions of the NAO soccer field (in mm).

The NAO robots used in the competition cannot be altered; changing a robot leads to disqualification. On software level, the only software provided by teams is the software for their own robots. This means that the main focus for every team is research: improving existing algorithms and developing new techniques and strategies.

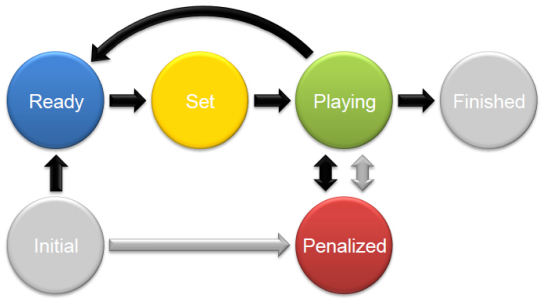

Figure 2: The different states the robots can be in during a soccer match.

The game is controlled by the Game Controller, a program that sends the current *state* of the game to the NAO robots. All the possible states are shown in Figure 2. Almost all states are used before the game commences; only the Playing and Penalized states are used during the game. The Playing state is the normal game state, where the robot just plays the game; the Penalized state is used in case a robot breaks a rule, or falls down. The robot should then be removed from the game, and is not allowed to move until it is removed from this state again. In addition to the Game Controller, human referees aid in watching over the game, helping to find errors caused by robots that result in Penalized state.

1.2 Research Questions

This project focuses on the following research questions:

1. Is it possible for a Nao to learn how to shoot the ball?
2. How can the NAO recognize the ball, the goals and the lines on the field?
3. What is the best image processing approach for NAOs?
4. Is it possible for a NAO to calibrate its vision automatically?
5. Can the NAO determine its position on the field?

1.3 Structure

The structure of this report is as follows. In this section, a short introduction is given on the challenges of robot soccer. It also includes the research questions that will be addressed in this paper, and a short structural overview of the paper. An overview of the official Robocup rules are stated here as well. Section 2 describes the environment in which this research is done, explaining the hardware and software used. In Section 3, all the physical actions the robot has to undertake to play soccer are described. The ways in which walking, aiming and kicking are handled is explained here. Real world noise, such as walking inaccuracy because of the surface, is also addressed here, as well as the technique used to resolve this issue. Section 4 is about image recognition, and splits into two subsections: recognizing the lines, and recognizing the ball and goal. The algorithms used for these problems are discussed. The implementation of these algorithms is described in Section 5. Section 6 is about future research, presenting possible research topics to continue with in the field of robot soccer. Section 7 contains a general conclusion that provides answers to the research questions, supplemented by general remarks and additional findings.

2 Environment

In this section, the hardware components as well as the technical specifications will be listed. An overview of the software that was used will be given after that.

2.1 Hardware

The NAO humanoid robot has been the standard for the RoboCup Standard Platform League since 2008. It is developed by the French company Aldebaran Robotics, founded in 2005.

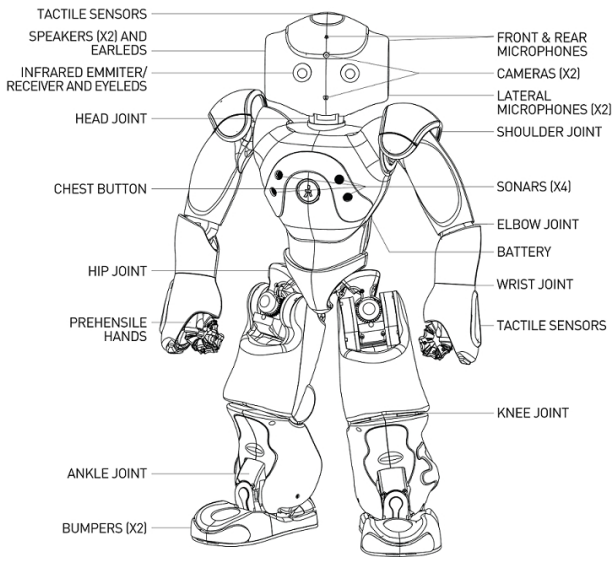

Figure 3: NAO joints and sensors.

All of the joints in Figure 3 can move independently of each other. The technical specifications of the NAO are listed in Table 1.

Technical Specifications	
Version	3.2
Body type	H25
Degrees of freedom	25
Height	573,2 mm
Weight	4,8 kg
Autonomy	60-90 min.
CPU	x86 AMD GEODE 500MHz CPU
Memory	256 MB SDRAM / 2 GB flash memory
Cameras	2 x VGA@30fps
Connectivity	Ethernet, Wi-Fi

Table 1: Technical Specifications.

2.2 Software

The NAOqi SDK is a cross-platform, cross-language programming framework in which all programs for the NAOs are written. The framework allows creating new modules intercommunicating with standard and/or custom modules, and loading these as programs onto the NAO. It is cross-platform because it is possible to run it on Windows, Mac or Linux. It is cross-language because it supports a wide range of programming languages. It is only possible to write local modules using C/C++ and Python. For remotely accessing and controlling the NAOs, however, NAOqi also supports .NET, Java, Matlab and Urbi. The NAOqi SDK version 1.14 was used in this research.

- **Programming language:** C/C++. This was the obvious choice to make, since C++ is described on the Aldebaran website as the 'most complete framework', and it is the only language that allows the writing of real-time code, making the software run much faster on the NAO. Furthermore, when using C++, there is no need for a wrapper for another language in order to make OpenCV (the image processing toolbox used for this project) work, since this is also written in C++.
- **Building/Cross-compilation:** qiBuild[2]. This cross-platform compiling tool, based on CMake, makes creating and building NAOqi projects easy by managing dependencies between projects and supporting cross-compilation (ability to build binary files executable on a platform different from the building platform).
- **Image processing:** Open Source Computer Vision (OpenCV). This is the image processing library that is supported by the NAOqi SDK. Version 2.3.1 was used in this research, since this is the latest version supported by the NAOqi SDK v1.14.
- **Higher-level robot control:** Choregraphe. Choregraphe is a graphical tool developed by Aldebaran Robotics that allows easy access to and control of the robot. From within Choregraphe, it is possible to control individual joints or create a sequence of existing modules to be executed by the robot.

3 Motion

In order to let the NAOs perform the difficult tasks that are involved in playing soccer, advanced motion control is a crucial part that has to be mastered. In this section basic movement for playing soccer is discussed.

Because enabling the NAO to move is such a difficult task and an important aspect, there has already been a lot of research in this area. A widely used approach, developed by Kajita et al[5], models balancing the robots as a 3-dimensional version of the linear inverted pendulum model, which they subsequently use to generate walking patterns for two-legged robots. This method, along with the other commonly used methods[11], is substantiated with a very extensive mathematical model. However, since the research questions and the overall focus of this research are on vision and image processing, these mathematical descriptions are considered out of the scope of this research and are omitted from the paper. A short overview of the three tasks incorporated within the developed algorithm is given instead.

The first part discusses moving around on the field (e.g. covering an area or approaching the ball). The second part covers positioning the robot in front of the ball in order to aim for a kick. The third and last part is about kicking the ball.

3.1 Walking

When playing soccer, being able to walk towards the ball is an essential task. Although this task is very straightforward for a human to perform, it gets more difficult when applying it to the NAOs. Humans move all their joints intuitively in order to walk, run, or move their arms, whereas NAO robots need an underlying algorithm to determine the correct joint configurations for every type of movement.

For walking, the standard walking algorithm developed by Aldebaran Robotics was used, the model of which is inspired by the previously mentioned 3D linear inverted pendulum model used in the article of Kajita et al[5]. The model is solved using quadratic programming as described by Wieber[13]. The implementation of this model is described in Section 5.1.

3.2 Aiming

When a robot is navigated in front of the ball, there are two options: either the ball is kicked with a certain angle, or the NAO positions itself behind the ball, in order to make a straight shot. In this research, the last option was chosen, since it is hard to determine the angle with which a ball is kicked. Although it is possible for the robot to position itself when approaching the ball, this method appeared to be too inaccurate for perfect positioning. So, when the robot approaches the ball, it stops in front of it, and then starts circling around until its shooting foot is in line with the ball and the target, a team mate or the goal. This circling around is done with multiple formulas for multiple cases. There are seven cases, 90, 180, 270 degrees, and the range between them. To clarify this Figure 4 is provided.

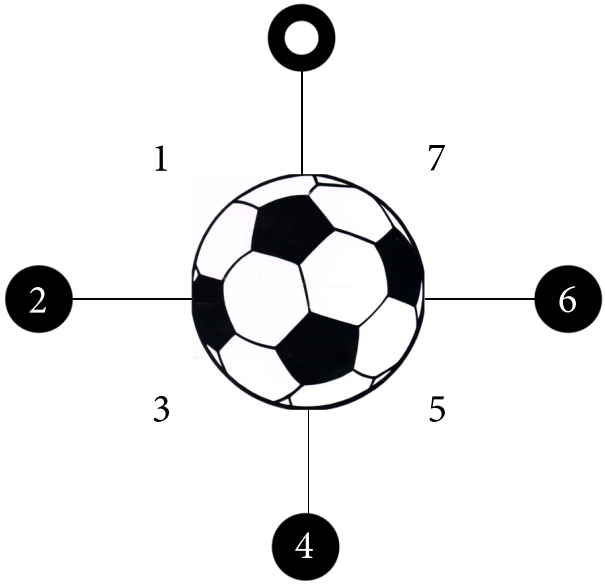

Figure 4: The different cases for aiming the ball.

When the distance to the ball is known, and angle that the NAO needs to turn, then the ending point can be calculated through geometrics. The list of formulas for these cases are shown in Equation 1, where x is the distance it has to travel on the x-axis, y is the distance it has to travel on the y-axis, r is the distance to the ball and α the angle.

$$
\begin{aligned}
&\textbf{Case 1}: x = \sin(\alpha) \cdot r \\
&\qquad\quad y = r\sin(\pi/2\alpha) \cdot r \\
&\textbf{Case 2}: x = -r \\
&\qquad\quad y = r \\
&\textbf{Case 3}: x = \sin(\alpha\pi/2) \cdot r \\
&\qquad\quad y = r + \sin(\alpha\pi/2) \cdot r \\
&\textbf{Case 4}: x = 0 \\
&\qquad\quad y = 2 \cdot r \\
&\textbf{Case 5}: x = \sin(-\alpha - \pi/2) \cdot r \\
&\qquad\quad y = r + \sin(\pi/2\alpha) \cdot r \\
&\textbf{Case 6}: x = r \\
&\qquad\quad y = r \\
&\textbf{Case 7}: x = \sin(-\alpha) \cdot r \\
&\qquad\quad y = r\sin(\pi/2 + \alpha) \cdot r
\end{aligned}
\tag{1}
$$

3.3 Kicking

When the robot has walked towards the ball, and it has gotten into position, then the last step is kicking the ball. Kicking is a complex motion, because it is very hard for a NAO robot to maintain balance while at the same time swing his leg hard enough to generate enough power. Fortunately, an existing list of joint movements describing a kick could be used[6].

4 Image Processing

Because NAO soccer consists of a lot of areas to focus/research on, each project on improving NAO soccer has an emphasis on a certain topic. For this project, the emphasis is mostly put on the next field, called image processing. The field of image processing focuses on analysing visual input given by the NAO. In this research, two aspects of NAO soccer were addressed, namely field recognition and ball recognition. Ball recognition is logically used for finding the ball, while field recognition is mainly for determining the robot's position on the field, also known as localization. Different algorithms used for these aspects of image recognition are described and explained.

4.1 Field Recognition

This part discusses three types of algorithms used for recognizing lines on the field in images captured by the NAO's camera. The Canny Edge algorithm, used for detecting edges (e.g. the corners of the field), will be explained first. Secondly, Hough line transform, a technique that is used for detecting straight lines, will be described. Lastly, Feature Detection will be discussed.

Field recognition is crucial when applying localization to a NAO robot's play. The NAO can map the lines that are found to a (known) global map of the field to find its current position on the field, in order to make sure that it is walking in the right direction.

Canny edge

Canny Edge is an algorithm developed by John F. Canny in 1986[3]. It is designed to be the optimal edge detector (according to some criteria). The Canny Edge algorithm tries to satisfy the following criteria:

- **Good detection:** Only true existent edges are detected and almost no fake edges are detected.
- **Good localization:** Minimizing the distance between the edge pixels detected and the real edge pixels.
- **Minimal response:** Only one detector gives a response for a single edge. Multiple edges should not be detected on a single edge.

The algorithm consists of four steps.

The first step is to filter out any noise using a low pass filter. Convolving the raw image with a two-dimensional kernel of Gaussian values, also known as *Gaussian blur*, is the most commonly used technique for this purpose. The size of the kernel determines the amount of blurring. An example of a 5×5 Gaussian filter, taken from the OpenCV documentation, is given in Equation 2, where $\boldsymbol{B}$ is the filtered image, $\boldsymbol{A}$ is the raw image, $*$ is the convolution operator, and the constant 159 is the sum of all the elements in the Gaussian kernel:

$$\boldsymbol{B} = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} * \boldsymbol{A} \tag{2}$$

The second step finds the intensity gradient of the image. In this step, Canny edge applies the *edge detection operator*: a pair of convolution masks in both x and y direction:

$$\begin{aligned} \boldsymbol{G}_x &= \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \\ \boldsymbol{G}_y &= \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \end{aligned} \tag{3}$$

The edge gradient $\boldsymbol{G}$ and direction $\boldsymbol{\Theta}$ can then be calculated from the first derivative in the horizontal direction $\boldsymbol{G}_x$ and the vertical direction $\boldsymbol{G}_y$ returned by the edge detection operator:

$$\begin{aligned} \boldsymbol{G} &= \sqrt{\boldsymbol{G}_x^2 + \boldsymbol{G}_y^2} \\ \boldsymbol{\Theta} &= \arctan \frac{\boldsymbol{G}_y}{\boldsymbol{G}_x} \end{aligned} \tag{4}$$

The direction angle of the edges are then rounded to the horizontal, vertical, or one of the two diagonal angles (0, 45, 90, and 135 degrees).

The third step consists of removing the pixels that are not considered to be part of an edge. Only the candidate edges will remain. The image is scanned along the image gradient direction, and if pixels are not part of the local maxima they are set to zero. This has the effect of suppressing all image information that is not part of local maxima. This technique is also known as non-maximum suppression.

In the last step, Canny edge determines whether the candidate edges should be accepted or not. To decide upon this, two thresholds are used (an upper and a lower one). The candidate selection procedure goes as follows, for all pixels:

- If the pixel gradient is higher than the upper threshold, the pixel is accepted as an edge.
- If the pixel gradient is below the lower threshold, it is rejected.
- If the pixel gradient is between the two thresholds, it is only accepted if it is connected to a pixel that is above the upper threshold.

Canny himself recommended using an upper to lower ratio between 2:1 and 3:1.

The soccer field can be seen as a flat space where the only edges are the edges between the lines and the green space in the field. Therefore, the Canny edge algorithm is able to detect the lines around the NAO. In this case, it actually works very well, except for the fact that Canny edge will detect all edges it can find. This means that white spots in the grass, the ball and other NAOs will be found as edges too. A good example of this can be seen in Figure 5. A possible solution is to increase the size of the Gaussian kernel in Equation 2, in order to increase the amount of blurring, making the white spots blend into the surroundings. Note that by doing this, we might also lose the desired corner points.

Another problem of Canny edge is that in a highly detailed, sharp picture, a single edge will be detected as multiple edges close to each other. The Canny edge algorithm does not allow for specifying how far two edges should be distanced from each other in order to be considered as separate edges. Although it is possible to work with the result of the Canny edge algorithm, the result will be inaccurate data, which can cause big noise when used for calculations such as determining the position of the NAO.

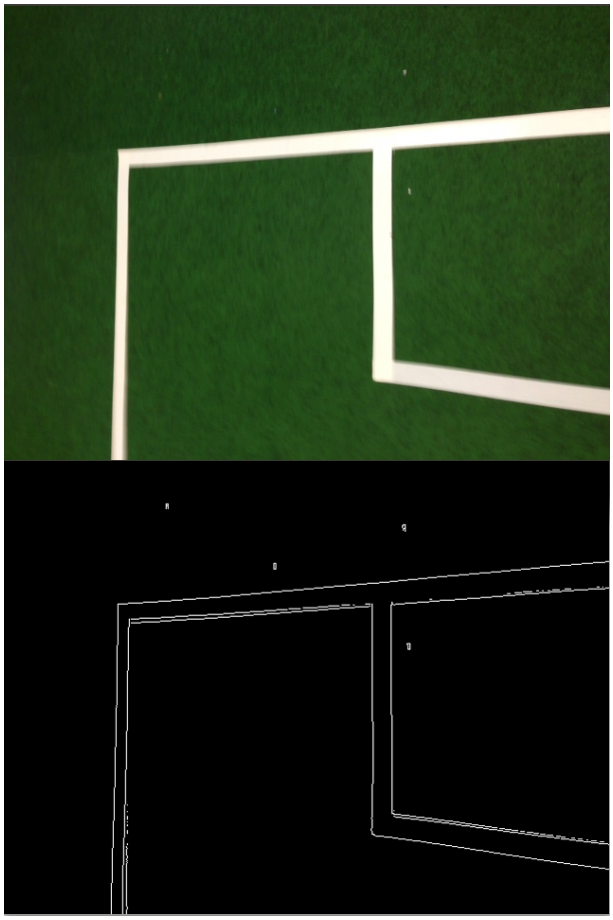

Figure 5: White spots in the field that are detected as edges.

Hough line transform

The Hough line transform is a transform that can be used to detect straight lines. It is recommended to pre-process the image with an edge detection operator (like Canny edge) before applying the transform. Hough line transform first creates a sinusoid for each point in the image. This sinusoid represents the family of lines that go through this point using the polar coordinate system. A line equation can be written in polar coordinates using Equation 5:

$$r = x \cos\theta + y \sin\theta \tag{5}$$

The family of lines going through a point (x_0, y_0) can be found by substituting x_0 and y_0 in Equation 6:

$$r_\theta = x_0 \cdot \cos\theta + y_0 \cdot \sin\theta \tag{6}$$

meaning that each pair r_θ, θ represents each line that passes by (x_0, y_0).

A simple example is provided in Figure 6. The four white dots on the left are several points in the polar coordinate frame. They each have a corresponding sinusoid in the Cartesian coordinate frame, shown on the right side of the figure. The white dot on the right is the intersection between these four sinusoids. As you can see, that exact intersection corresponds to the line going right through all four points in the polar coordinate frame. The formula of this line is then determined by a line that comes from the origin (the center) of the image, and is perpendicular to the line.

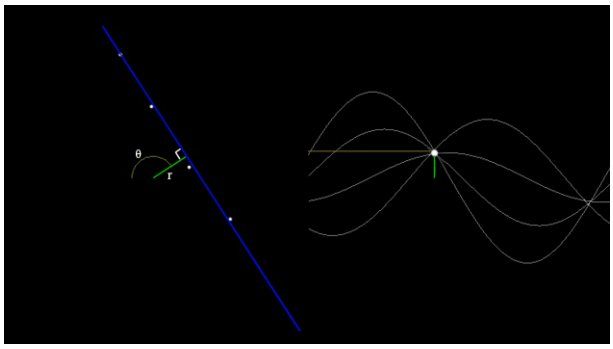

Figure 6: A visualization of the various sinusoids in the Cartesian coordinate frame corresponding to the white dots in the polar coordinate frame.

By applying Equation 6 to all points, (such that $r > 0$ and $0 < \theta < 2\pi$), we get a number of sinusoids equal to the number of points in the picture. If the curves of two different points intersect in the plane (θ, r), that means that both points belong to the same line. The result of this is that we can find straight lines by looking at the intersections of all the drawn curves. The more curves that intersect in one point, the more points there are on the line represented by that intersection. A threshold can be set to define the minimal number of intersections needed to detect a line. Hough line transform keeps track of the intersection between curves of every point in the image. If the number of intersections is above some threshold, it declares it as a line with parameters (θ, r) of the intersection point.

The Hough line transform has two implementations. The probabilistic version of the Hough line transform is a more efficient implementation for the algorithm described above. The output from the probabilistic Hough line transform differs; The Standard Hough line transform returns the lines in polar coordinates, whereas the Probabilistic Hough line transform returns the start and end point of each detected line in Cartesian coordinates.

As we are looking for lines, Hough line transform gives the best result for detecting the lines of the soc-

cer field. As mentioned before, Canny Edge performs okay, but as it looks for edges it usually finds also other edges of non-line objects. Applying Hough line transform after pre-processing the image (with Canny Edge, for example) filters out all the edges which are not lines. An example of this can be seen in Figure 7, where the small edges found by Canny Edge are not detected as lines.

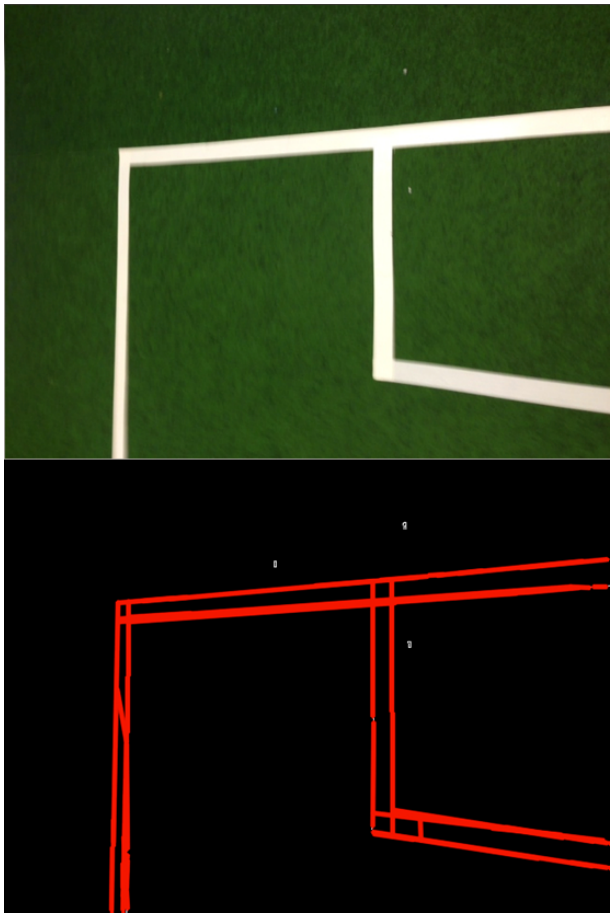

Figure 7: The white spots are not detected as lines.

Feature detection

Another way to analyze images is to split the image into certain *features*. In this context, features are defined as uniquely recognizable characteristics of an image. Frequently occurring features in an image are edges, corners or blobs (also known as regions of interest). In this case, feature detection is used to detect corners in an image.

Because a corner is the intersection of two edges, it represents a point in which the directions of these two edges change. Hence, the gradient of the image has a high variation in this point. Corner detection is based on this principle.

The algorithm starts by finding the most prominent corners in the image or in the specified image region. A *quality measure* is then calculated for every pixel in the region, using either a minimal eigenvalue or a Harris corner detector approach:

- **Minimal eigenvalue:** For every pixel p, a $n \times n$ neighborhood $S(p)$, where n is the block size, is considered. The covariance matrix of the derivatives is then formed over the neighborhood[9]. After that, the eigenvectors and eigenvalues of this matrix are calculated and only the minimal eigenvalues of these matrices are stored. These values are then used as the corner quality measure.
- **Harris corner detector:** A 2×2 gradient covariance matrix over an $n \times n$ neighborhood (n again being the block size) is calculated for each pixel. The corner quality measure is then calculated.

After this the function performs a non-maximum suppression, as described in the Canny edge algorithm. The local maxima in a 3×3 neighborhood are found. All the corners with a minimal eigenvalue less then the quality measure, multiplied by the quality measure of that corner, are then discarded. The remaining corners are sorted by quality measure in descending order. Each corner for which there is a stronger corner at a distance less than a certain threshold is then discarded.

As the goal is to recognize the environment around the NAO, this can also be determined by looking at all the corner points around us. Corner points represent intersection of lines, and by finding all the corner points, it is possible to manually draw lines between each of the corner points, to find the full environment around us.

4.2 Ball Recognition

Ball recognition focuses on recognizing the orange soccer ball used in NAO soccer. This can be done by using algorithms designed to detect circles. The three techniques that are used are the Hough circle transform, contour finding, and moments. As a positive side note, these algorithms can also be used to recognize the goals, with only minor adjustments.

Hough circle transform

Hough circle transform[7] works roughly the same as the Hough line transform mentioned in Section 4.1. The reason it is only roughly the same is that an extra dimension would have to be added to the line transform. Recall from Section 4.1 that a line can be defined in

polar coordinates using the variables (r, θ). To define a circle, however, three parameters have to be used: (x, y, r), where (x, y) is the center of the circle and r the radius. This extra dimension in the definition of a circle means far greater memory requirements and much slower speed when using the same operations described in Section 4.1. A way to avoid this is by using a more efficient method called the Hough gradient method.

The Hough gradient method works as follows: first, the image is pre-processed through an edge detector, such as Canny Edge or any other edge detector. Next, for every nonzero point in this pre-processed image, the local gradient is considered. Using this gradient, every point along the line indicated by this slope is added to the family of sinusoids representing this line. At the same time, the location of every one of these nonzero pixels in the edge image is noted. The candidate centers are then selected from those points in the family of sinusoids that are both above some given threshold, and larger than all of their immediate neighbors. These candidate centers are sorted in descending order of their values, so that the centers with the most supporting pixels appear first. Next, for each center, all of the nonzero pixels are considered. These pixels are sorted according to their distance from the center. Working out from the smallest distances to the maximum radius, a single radius is selected that is best supported by the nonzero pixels. A center is chosen if it has sufficient support from the nonzero pixels in the pre-processed image and if it is a sufficient distance from any previously selected center.

Contour finding

The second technique for finding the ball is a technique which tries to detect the contours of the ball. This is done by converting a image first into a so called binary picture, which is a picture with only black and white (0 and 1 in mathematical terms) pixels. This is done by filtering out all the colours that are within a certain range of hue, saturation and value in the HSV colour spectrum. A nonzero point in the image is then picked, and, using the border following algorithm by Suzuki[12], the border of this nonzero area is tracked. This algorithm distinguishes between inner and outer borders, so circular shapes are well-suited because of their characteristic shape. The contour, the outer line of the ball, is found as a result. In-depth mathematical details on this algorithm can be found in the referred paper.

The coordinates of the center of the ball are crucial for distance calculations performed by the robot, and these are not provided by the contour finding algorithm. To solve this problem, the algorithm can be combined with the technique described below.

Moments

This technique uses the mathematical concept of moments to calculate so called image moments. An image moment is defined as "a weighted average of the image pixels' intensities, or a function of such moments, usually chosen to have some attractive property or interpretation"[4]. In the case of ball detection, it is used to find the geometric center of the ball, but also its radius and area, so the actual properties of the ball. The results from the contour finding are used as an input here, because it gives the exact points needed to calculate these properties. Moments can be calculated using Equation 7:

$$M_{ij} = \sum_x \sum_y x^i y^i \cdot I(x, y) \tag{7}$$

The i and j values determine the order of the moment, the x and y and $I(x, y)$ are the coordinates of the points and the intensity of these coordinates, respectively, given by the contour finding algorithm. When i and j are equal to zero, the area of the ball is calculated. This means, using the properties of a circle, the radius of the circle can be found using Equation 8:

$$r = \sqrt{\frac{M_{00}}{\pi}} \tag{8}$$

Furthermore, the coordinates of the center of the ball can be calculated fairly easily as well. This is done as shown in Formula 9:

$$\begin{aligned} x &= \frac{M_{10}}{M_{00}} \\ y &= \frac{M_{01}}{M_{00}} \end{aligned} \tag{9}$$

Using these three equations, the desired properties of the circle can be found without diving into the mathematically challenging aspects of moments.

5 Implementation

This section will discuss the implementation of the NAO robot and the different tasks it has to perform. As mentioned in Section 2.2, the NAOqi SDK version 1.14 was used, and the modules were written in C/C++.
Section 5.1 will discuss the implementation of the motion tasks, and Section 5.2 is about the way image processing was implemented in this research.

5.1 Motion

As described in Section 3, the motion problem for the NAO can be split up in three different tasks: walking, aiming, and kicking.

Walking

The NAOqi SDK has a dedicated API, called *Locomotion Control*[1], that includes all the methods necessary to make the NAOs walk. This API contains methods that simulate human walking based on the previously mentioned model of Kajita et al[5], solved using quadratic programming[13]. The two main methods for making the NAO walk are *move()* and *moveTo()*:

- **ALMotionProxy::move(const float& *x*, const float& *y*, const float& *theta*):** Makes the NAO move at the given velocity, where the velocity is determined by the length of the vector xy, where the maximum velocity equals the unit vector, and *theta* is the rotation around the z-axis in radians. If the length of xy is smaller than 1, the velocity is the fraction of the unit vector; if the length of xy is bigger than 1, the vector is normalized. This walk is endless, and can only be terminated using another method like *stopMove()*, or any interrupting event like picking up the NAO or falling down. This is a non-blocking call, meaning that during the execution of this method, it is possible to perform parallel tasks like scanning the field for the ball.
- **ALMotionProxy::moveTo(const float& *x*, const float& *y*, const float& *theta*):** Makes the NAO move to a given position relative to the robot, expressed in distance along the x- and y-axis, and the rotation around the z-axis in radians. The walk is terminated as soon as the NAO has traversed the given trajectory. This is a blocking-call, meaning that it is not possible to perform tasks in parallel while executing this method.

For this research, a custom module was created, containing wrapper/convenience methods for turning the NAO, walking straight, and making the NAO stop walking either smoothly or abruptly. Also, the angles are expressed in degrees instead of radians. The documentation of this module can be found in the appendices.

The NAOs walk is stabilized using feedback from its joint sensors. This makes the walk robust against small disturbances and absorbs torso oscillations. A beneficial side-effect of this is that the camera in the head of the NAO will stay pretty stable, making image processing while walking easier.

In reality, the actual path the NAO traverses deviates from the desired path. The amount of deviation depends on the surface the NAO is on. If the deviation is a fixed angle, this problem is easily solvable: correct the deviation by subtracting an angle equal to the deviation from the desired angle. This works for short distances; when walking long distances, however, it becomes clear that the deviation is not a fixed angle. A better solution to this problem is to use the non-blocking *move()* method for long distances, and simultaneously use image processing to check whether the ball is still right in front of the NAO. When the path deviates, adjust the angle such that the ball gets in front of the NAO again.

Aiming

The same custom walking module that was used for walking contains a method *walkAroundBall()*, which uses the calculations from Section 3 to determine the distance and angle that is needed to turn around the ball properly.

- **Motion::walkAroundBall(const float& *thetaInDegrees*, const float& *distanceToBall*):** Calculates the coordinates and rotation neccesary to turn around the ball with the desired angle *thetaInDegrees*. For *distanceToBall*, 20 is a good estimate.
- **Motion::walkAroundBallInSteps (const float& *thetaInDegrees*, const float& *distanceToBall*, const int *inSteps*):** Divides walking around the ball into a number of steps in order to decrease the error due to inaccuracy.

Kicking

A custom kick module was created, containing a kick method that uses a list of joint coordinates from a master project at the Department of Knowledge Engineering at Maastricht University[6]:

- **Kick::doKick():** Executes the list of joint movements necessary for a proper kick.

The documentation of this module is also provided in the appendices.

5.2 Image Processing

As mentioned in Section 2.2, OpenCV 2.4.3 was used for image processing, mainly because all of the algorithms discussed in this paper are already implemented as OpenCV functions. The libraries are written in C/C++ and are well-documented[8].

Canny edge

In OpenCV, the *Canny()* method is a complete implementation of the Canny edge algorithm. It returns an edge map containing all of the edges found in the input image. Additionally, the method is provided with two parameters for an upper and a lower threshold for the candidate selection step.

Hough line transform
The method *HoughLines()* returns a vector of lines represented in polar coordinates, found in the input image after applying the Hough line transform on it. A threshold also has to be specified for the number of points that belong to a line in order for it to be considered as a line. The method *HoughLinesP()* finds line segments using the probabilistic Hough transform. The probabilistic Hough line transform returns a vector of lines, each line represented by a vector (x_1, y_1, x_2, y_2) which represents the two end points of a line.

Feature detection
The method *goodFeaturesToTrack()* returns a specified number of corners found in an image. This number is given as a parameter in the method. If the number of corners found is greater than the desired numbers, it only returns the four corners that are the most prominent.

Hough circle transform
HoughCircles() finds circles in a preprocessed image using the Hough circle transform, The circles are represented as (x, y, r).

Contour finding
There is a method, called *findContours()*, that finds all the borders in an image and returns them. This is not restricted to circles; in order to do that, the contours must be processed by moments.

Moments
The *Moments()* method takes the contours of an images, calculates the moments up to the third order of the contours, and stores them as an object *Moment*. From that, the center of the circles in the image can be calculated by taking the first three moments in the image (m_{00}, m_{01}, m_{10}). Dividing m_{10} by m_{00} provides the x-coordinate of the center of the circle; dividing m_{01} by m_{00}) will give the y-coordinate; the calculation $\sqrt{\frac{m_{00}}{\pi}}$ gives the radius of the circle.

5.3 Algorithms

In this section, a description of the algorithms for finding a ball and finding a goal are given.

Finding the ball
As soon as a snapshot of the camera of the NAO is taken, it is transformed from RGB into the HSV format. The R, G, and B components of an objects color in the original frame are all correlated with the amount of light shining on the object, and are therefore correlated with each other. This makes it very hard to distinguish the object from the rest of the frame in terms of those components. Defining an image in terms of Hue, Saturation and Value (HSV) separates the intensity of the image from the color information, which allows for a better description of an object in the image.

After that, a mask of the frame is created, using only black and white as colors. First, a range of HSV values of the colors that we want to find is specified. Each pixel is then evaluated in the HSV frame and checked whether it is in our specified HSV range. If the pixel is in range, a white pixel will be drawn on the mask; if not, a black pixel will be drawn. The mask now only contains white and black pixels and there is a good contrast between the pixels of interest and the pixels not of interest.

Next, the contour-finding algorithm is used to find the contours of groups of white pixels in the mask. By doing this, any noise coming from white pixels separated from the bigger groups of white pixels is filtered out. Filtering noise in this way is valid in this case, because it is known that the object of interest must consist of multiple pixels in the mask. After running the algorithm the locations of the contours of the groups of white pixels are known, and these are the regions of interest for the next step.

This step consists of discerning circular shapes among the groups of pixels that were found by the contour-finding algorithm. This is done by calculating the moments of the shapes, which was discussed in Section 4.2. Before calculate the moments, some eroding and dilating is applied to the contours. This means that the contours are transformed a bit more into a circular shape, such that the moment algorithm has a better chance of detecting a circle among the contours. All the circles found are returned after scanning each of the contours, and the algorithm is finished.

Calculating the distance to the ball
After focusing the NAOs head to the ball, such that the ball is in the center of the NAOs vision, the NAO's body has to turn into the direction the head turned to. Since the NAO is now staring straight to the ball, calculations can be made to locate the coordinates of the ball. First of all, the height of the camera is known, as well as the angle in which the head is tilted. Because only the distance to the ball has to be calculated (since the NAO is standing perfectly in line with the ball), this is enough information to apply Equation 10:

$$\hat{d} = \frac{h}{\tan(\alpha)} \tag{10}$$

where $\hat{d}$ is the estimated distance to the ball, h is the height of the camera and α is the angle in which the head is tilted.

With $\hat{d}$ known, we can let the NAO walk to the position of the ball. Since the estimated distance assumes that the ball is a tiny point on the ground, instead of a ball of which the center is a few centimeters from the ground (see Figure 8), a certain distance has to be subtracted from $\hat{d}$.

$$\begin{aligned} \epsilon &= \frac{r}{\alpha} \\ d &= \hat{d} - \epsilon - 2r \end{aligned} \tag{11}$$

where ϵ is the distance from the ground to the center of the ball, r is the radius from the ball and d is the actual distance from the NAO to the ball. r is subtracted twice from $\hat{d}$, so that the NAO ends up in front of the ball instead of bumping into the ball.

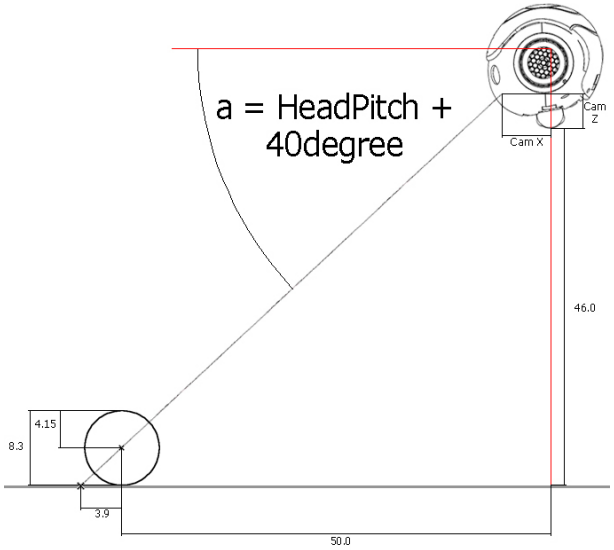

Figure 8: The NAO sees 'through' the ball.

Finding the goal

Recognizing the goal differs slightly from recognizing the ball. The first thing that needs to be done is cutting of everything above the field. In this way it is certain that there is no noise in the background. After doing this, the two poles of the goal have to be found. The poles are ether colored blue or yellow. Two masks are created and the union is taken of this to get a mask that has the yellow and blue posts in it. The contour finding algorithm is now used to find all the contours of the blobs in the mask. The difference with ball recognition is that there is no need for moments to detect circles. The contours that are found already represent the posts.

5.4 Strategy

The state diagram provided in the appendices describes the current strategy of each robot, how it deals with exceptions and how it will behave.

The first step in the state diagram is the initialization. The robots will be placed on the soccer field and are not aware of each other. They do not know where the ball and the goals are yet and will need to figure this out first.

The second step is finding the ball. The searching module will be activated which will have the NAO look around for the ball. It will eventually check the whole space 360 degrees around him to find the ball. The assumption can therefore be made that it will find the ball if it did not move in the meantime. If it did, it will continue performing the searching module until it finds the ball.

Once it finds the ball, the approximate position of the ball will be calculated. An infinite walk will now be initiated towards the position of the ball, while in the meantime the NAO keeps checking for the ball, and keeps recalculating the position of the ball. This means that if the ball moves slightly to the right, the NAO will adapt its movement and keep walking towards it. This also nullifies the fact that the NAO will not completely walk in a straight line, as the NAO will adapt its angle each time after recalculating the position of the ball. If the ball somehow gets kicked away by another robot or gets taken away completely from the view of the NAO, it will stop and reactivate the searching module to find the ball again.

Once it has reached the ball, the next step is to determine the position of one of the goals. Finding one of the goals is sufficient, as the dimensions of the field are known and the position of the other goal can be calculated the position of a single goal. Once the position of the opponent goals has been found, the angle that is necessary to turn can be calculated, in order to shoot the ball in the direction of the goal.

The rotating module is now activated to rotate around the ball and put the right foot of the NAO in the correct position.

Once this is done it is time to run the kicking module. This module will perform the kick with a certain power. The distance between our position and the goal could now be calculated to determine the power of the kick. In practice, however, kicking the ball at maximum power makes the ball move better in a straight line and the kick is more accurate. Therefore, the NAO always shoots with maximum power.

Once the ball is shot, the Game Controller will notify the NAO whether a goal has been made or not. In both cases, the NAO will start again at the first state to find the ball again, and will continue from there.

6 Future Research

There are two major areas that investigation into would be logical to continue with.

The first area is localization. Localization is the practical use of the field recognition that was created. Though it has not been researched right now, a fundamental part of the tools to build it are clearly already here. The next step would be to normalize the lines found, and then map them onto an internal map, to find the possible positions of the NAO. Then, the exact location can be found by also taking into account the location of the goal, and its colour. This is a complex problem and could easily be a main focus for a new project.

The second area is running modules internally. A lot of time and research has been put into this during this project, since it is a terrain where almost no research has been done in before. This has been completed partially, but the part that did not function yet internally was the image recognition. This is important, because the frame rate of the camera would increase dramatically. Through Wi-fi, the camera only has a speed of 2 frames per second, and through cable connection it has a speed of around 12 fps. But when the image recognition runs internally, it could easily reach a framerate between 20 and 30, which would mean that it (almost) has fluent vision. This would enable the NAO to follow a moving ball, improving ball tracking a lot.

7 Conclusion

Various aspects of NAO soccer have been investigated, and the following conclusions can be drawn from this. First, the research questions from the introduction are answered.

The first question stated was: Is it possible for a NAO to learn how to shoot the ball? The NAO can undeniably shoot the ball by using a list of various joint movements.

The next question was if it was possible for the NAO to recognize the ball, the goals and the lines on the field. This has been found true as well, though there is a difference in techniques used for recognizing these objects. The techniques used for recognizing the goal and ball were the Hough circle transform, contour finding and moments. The techniques used for field recognition are Canny edge, Hough line transform and feature detection. The distinction between the two kinds of image recognition is mainly because the field recognition has an different goal from recognizing the other two objects. Field recognition is mainly used for enabling a NAO to locate itself on the field.

The next research question was which algorithm was best for image processing. This answer has to be split up into the two subsections, described in the answer of the last research question. For the ball recognition, no specific best approach can be chosen, though through practice it was found out that combining the contour finding and moments approach works decent and stable in finding the ball and the goal. For field recognition, all the techniques have found to be working, but none of them really stood out. If one has to be chosen, then it would be the Hough line transform, since it handled the noise better.

The next question that has to be answered is whether it is possible for a NAO to calibrate its vision automatically. The tests that were performed on this topic have shown that it is possible to let a robot calibrate its parameters in such a way that it can recognize a ball in the light conditions of that particular situation.

The last question is whether it is possible for a NAO to determine its position on the field. This answer cannot be fully answered from this research, since localization has not been fully researched. The prequel for localization is field recognition, and though this has been researched, mapping the found lines onto an internal map to determine the position on the field has not been looked into.

The general impression on NAO soccer derived from this report is that though there has already been a lot of research on this topic, it is still far from ideal for performing real soccer matches. The physical aspects like walking deviation and instability make it very hard to perform researched topics really in practice, but the general complexity of NAO soccer is a problem. Using real strategies, recognizing other robots and other complex issues are still far from solved. So, if the RoboCup wants to win against the best human soccer team by 2050, they should really get a move on and continue researching.

References

[1] Aldebaran Robotics (2013a). Locomotion control. http://www.aldebaran-robotics.com/documentation.

[2] Aldebaran Robotics (2013b). qibuild 1.14 documentation. http://www.aldebaran-robotics.com/documentation.

[3] Canny, John F. (1986). A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

[4] Flusser, Jan and Suk, Toms (1999). On the calculation of image moments. *Pattern Recognition Letters*.

[5] Kajita, Shuuji (2001). The 3d linear inverted pendulum mode: a simple modeling for a biped walking pattern generation. *IEEE/RSJ International Conference on Intelligent Robots and Systems*.

[6] Kirstein, Sabrina, Kppers, Bastian, and Vadakkeparambil, Raphael (2012). Robot soccer. M.Sc. thesis, Maastricht University, Department of Knowledge Engineering.

[7] OpenCV Development Team (2012a). Hough circle transform. http://docs.opencv.org.

[8] OpenCV Development Team (2012b). Opencv 2.4.3 documentation. http://docs.opencv.org.

[9] Prokhorov, Alexander V. (2011). Covariance matrix. http://www.encyclopediaofmath.org.

[10] RoboCup Technical Committee (2012). Robocup standard platform league (nao) rule book. http://www.tzi.de/spl.

[11] Shamsuddin, Syamimi, Ismail, Luthffi Idzhar, Yussof, Hanafiah, Zahari, Nur Ismarrubie, Bahari, Saiful, Hashim, Hafizan, and Jaffar, Ahmed (2011). Humanoid robot nao: Review of control and motion exploration. *IEEE International Conference on Control System, Computing and Engineering*.

[12] Suzuki, Satoshi and Abe, Keiichi (1985). Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*.

[13] Wieber, Pierre-Brice (2006). Trajectory free linear model predictive control for stable walking in the presence of strong perturbations. *IEEE-RAS International Conference on Humanoid Robots*.