Feature Detection and Localization for the RoboCup Soccer SPL

Amogh Gudi *(23408036)*, Patrick de Kok *(5640318)*, Georgios K. Methenitis *(10407537)*, Nikolaas Steenbergen *(10333681)*

University of Amsterdam

February 5, 2013

In this project we propose a basic code for the RoboCup Soccer Standard Platform League for the Dutch Nao Team. We focused on the problem of visual feature recognition and the localization of the robot in a tournament setup. This project contains a detailed description of our implementation together with a conclusion and a brief evaluation. Since this report only covers feature detection and localization we give an outlook on what parts are to be implemented in addition to make the robots play soccer.

1 Introduction

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

– RoboCup Initiative, objective statement [**RoboCupObjective**, **kitano1997robocup**]

RoboCup encompasses a set of competitions, stimulating innovation and research towards reaching one of the grand challenges within both the robotics and AI community – defeating the most recent World Cup winning soccer team by a team of robots in 2050. This demands for systems which can deal with a dynamic environment with a continuous, incompletely observable state space, with non-symbolic sensor readings and with cooperation.

The best known competition of RoboCup is RoboCup Soccer. This competition encompasses 5 leagues, each with different rules imposed on the game as well as on the participating robots [**RoboCupSPL**]. These differences induce a different research focus within each league. One of those leagues is the Standard Platform League, where all teams use identical robots. The Robocup Federation states that "[t]herefore the teams concentrate on software development only, while still using state-of-the-art robots. Omnidirectional vision is not allowed, forcing decision-making to trade vision resources for self-localization and ball localization." [**RoboCupSPL**]

To keep on pushing research forwards and the community of participating teams motivated to implement and design new methods, the RoboCup Soccer competition updates the rules each year. One of the more recent updates in the rules is that all goal posts should be yellow, whereas before there were two differently colored goal posts. This puts an emphasis on the implementation and enhancement of localization algorithms as the field is completely symmetrical.

The Dutch Nao Team has participated since 2010 in the RoboCup Standard Platform League (SPL) [**DNT-TD10**, **DNT-TD11**, **DNT-TD12**] and plans doing so for 2013 as well [**DNT-TD13**]. The software package used during the previous cup has been written in Python. Its localization algorithm was very heuristic and course-grained; it did not made explicit where the robot was placed, but indicated whether the robot should kick the ball[1]. Currently, the Dutch Nao Team is transitioning its code from Python to C++ for speed gains.

The subject of this report can be split in two parts. Section 3 describes the first part, where we have implemented modules which extract several visual features located on the field. We describe which features are extracted and how the images coming from the NAO's camera are processed.

These features and their locations with respect to the robot are passed to the localization module from section 4. We first give a general description of the algorithm (i.e. Monte Carlo Localization) in section 4.1, and describe how the sensory information is processed and modeled (section 4.3 and 4.2). Section 4.4 and section 4.5 describe extensions implemented to the standard Monte Carlo Localization to improve the performance of the system on common problems in a RoboCup Soccer tournament situation. After the description we give a short evaluation of those approaches in section 5 and a conclusion in section 7. In the last section we will sketch future work, that needs to be done to have a functioning robot soccer team in a tournament situation.

2 Related Work

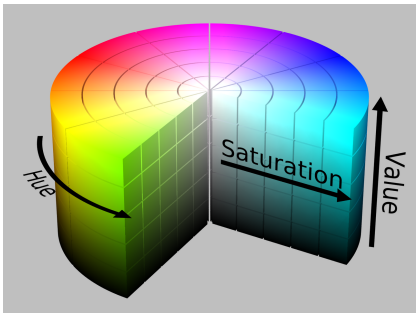
Canas et al. [**canas2009visual**] propose algorithms to detect only the goal from the robot's camera based on geometrical planes and based on color and edge filters and Hough transformations. However, these algorithms make the assumption that the goal is completely visible to the robot, but we know that this is not the case in most of the situations. This approach self localizes the robot only based on goal detection, which we might say is not a robust way.

Schulz and Behnke [**schulz2012utilizing**] give a method for line-based localization. This method recovers lines from images and dechyphers features from it, which are then used by localization algoritms like Monte-Carlo localization. This method appears to reduce dependance on colour-coded robot soccer environment.

Deng [**deng2011natural**] explores triangulation techniques on the detected features on the robot's horizon. However, although theoretically robust, this technique fails in the practical implementation because of noise in the image caused by movement.

[1] The robot would kick the ball in the direction of a goal post when in sight. The behaviour has not been changed since the field has been made symmetrical.

Figure 1: *HSV colorspace. (Image source)*



Ashar et al. [**ashar2010robocup**] present a well tested hierarchy of modules implementing perception, world modelling and behaviour generation, incorporating a simpler self-localization system. Due to time constraints we limited ourselves to feature detection and localization.

3 Feature Extraction

Visual object recognition is a key ability for autonomous robotic agents, especially in dynamic and partially observable environments. A reliable landmark detection process is really crucial for achieving self-localization, which can be easily considered as the stepping stone for having a functional robot soccer team. In this section, we describe the whole procedure of processing input images from the NAO's camera, outputting features that are informative to be used by the core of our localization system. Using a variety of image processing techniques, we successfully detect field landmarks. In general, we have to deal with noisy images, which not only contain the field region but also background noise and useless information which comes from above the horizon of the field. Furthermore, we had to deal with lighting conditions which may vary to a great extent in different places for different times. Also, the real-time constraints which had to be taken into consideration during our algorithmic implementation in order for the whole procedure to be able to execute in real time.

3.1 Choosing colorspace

The first challenge we came along was to choose the best colorspace representation in order to overcome the variations in lighting conditions. The dataset we had consisted of RGB images. Unfortunately, it is unfeasible to use the default RGB colorspace for most applications due to influences such as lighting and shadows. We also experimented with normalized RGB colorspace. Theoretically, normalized RGB would have given us the pure chromaticity values of each object in our field of view, and thus overcoming the problems shadows and lighting conditions can give. However, normalized RGB did not help us achieving a light-invariant color representation, resulting into false positives in color segmentation. HSV was the solution to our problem. HSV is a cylindrical colorspace representation, which contains information about hue, saturation, and value. We found hue to be really informative in order to distinguish easily between colors. Hue defines the pure chromaticity of the color and it is independent of the lighting conditions. In figure 1, this cylindrical color representation is illustrated. One can see that the hue is represented by the angular dimension in the vertical axis of this cylinder.

Table 1: *Threshold values used for color segmentation.*

	Threshold		
Color	**Hue**	**Saturation**	**Value**
Green	$20 \sim 37$	$100 \sim 255$	$100 \sim 255$
Yellow	$38 \sim 75$	$50 \sim 255$	$50 \sim 255$
White	$0 \sim 255$	$0 \sim 60$	$200 \sim 255$

3.2 Basic Pipeline

Having described the first basic step of our approach, choosing the proper colorspace for our application, it is time to go deeper into the feature extraction process. The first step in this process is the image input from the NAO's camera in HSV format. As we said before, images not only contain the important region of the field in which we are interested in, but also some background information which is useless in order to detect field features. This background usually extends above the field horizon and it may become really disturbing in the extraction process. This is the reason why background removal is the first step in our feature extraction scheme. Field lines, goals, ball, and other NAO robots are the features located into the field. In this project, we only took into consideration lines and the goals which are static landmarks, unlike the ball and other robots that are moving as we could not depend on them for self-localization.

The HSV values are used to binarize the image, with respect to the goals, and with respect to the lines. As we said in the introduction, the field is colored green, both goals are yellow, and lines are white. The next step of the feature extraction is the goal detection, making use of the horizontal and vertical histograms of yellow values. Line detection is followed in order to find a good estimation about the lines detected in our field of view. Having these lines' estimations, we can detect feature on them, using geometric properties. Last step in this process' pipeline is to output all these detected features to the localization core process. We can now enumerate all steps in these pipeline. These steps are:

1. Input HSV image from camera
2. Background removal
3. Image binarization
4. Line detection
5. Line feature detection
6. Goal detection
7. Output features

3.3 Image format

NAO's camera can output images in different colorspace representations and resolutions [**NaoCam**]. The images from the dataset we had were in RGB format, and they had QVGA resolution (320×240). We wanted to keep a low resolution in order to keep the time complexity of our algorithm feasible for real-time execution.

3.4 Background Removal

Background removal defines the task of detecting the field's borders in order to exclude uninformative regions in the processing image. By doing so, computation time is saved as only that part of the image is subtracted, where all the features are located

we are interested in. This can be done by a vertical scan of the image and detecting the first green pixel in each column. This method can work efficiently but it is not robust in many cases, as green pixels can be found above the field's horizon as well. Following the same principle in our approach, we consider as background every region in a column before a considerable amount of green pixels. We start in each column considering the pixel in the first row as background. Then, during scanning each column, we stop when we find a set amount of continuous pixels which are green and assign as horizon row in this column the first of this sequence of green pixels. Table 1 displays the hue, saturation and value thresholds for pixels to be classified as green. Figure 2 illustrates the process of background removal. Even though it is not a sophisticated method, we can see that in this specific example, almost all background above the field's horizon has been removed, helping us in this way to avoid detect false positive features in the background.

3.5 Image Binarization

In this section, we will talk about another image pre-processing technique which helped our main procedure detect features easier. Image binarization is a process which outputs two binary images, one in respect to the goals, and the other in respect to the field lines. For goal detection, it is natural that we are interested only in yellow areas of the image. The same applies for field lines, we are only interested in green and white areas of it. The main goal of this process is to classify colors according to table 1. As we can realize, a simple way to classify colors is used in order to get a binary image, with the interesting part highlighted in each case. For the first case, goals, we are only interested to find yellow, so we output white for each yellow pixel, and black for any other color. For field lines, black for every other than white pixel.

Figure 3: *Binary images in respect to the lines (left), and to the goal (right), for the example RGB image in Figure 2.*

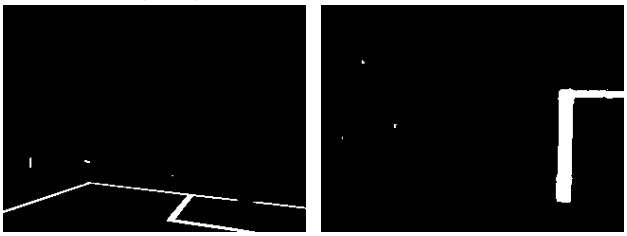


Figure 3, illustrates the process of image binarization. To add some technical details about the above implementation of image binarization, we can say that the whole process of it is integrated into the background removal procedure which requires only one vertical scan of the input image, and its goal is to minimize the space complexity of the whole feature extraction method, as now we have only to process binary images with only one channel, in contrast to the original 3-channel image.

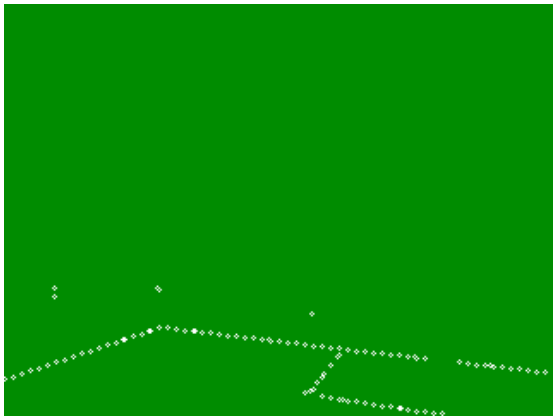
3.6 Line Detection

Before proceeding to the exact procedure about finding goal posts and line features, it is time to introduce our approach to detect line segments in a binary image like those we presented before in section 3.5.

3.6.1 Other approaches

In general, we are interested in finding one line segment for each actual line appearing in the image. Other methods, like Hough Transformation or Probabilistic Hough Transformation failed to detect these continuous line segments, due to the fact that they applied after edge detection step. Then, when edge detection is applied to those lines finds edges at both sides of an actual line. Therefore, both Hough transformations find lines at both sides of an actual line. There were also some other problems with these methods, namely, the big number of detected lines, small line pieces on each actual line, and wrongly connected line segments. Furthermore, the huge number of detected lines, made our efforts to cluster those lines tough.

Figure 4: *Points produced by black-white-black transitions in the example binary image of figure 3.*



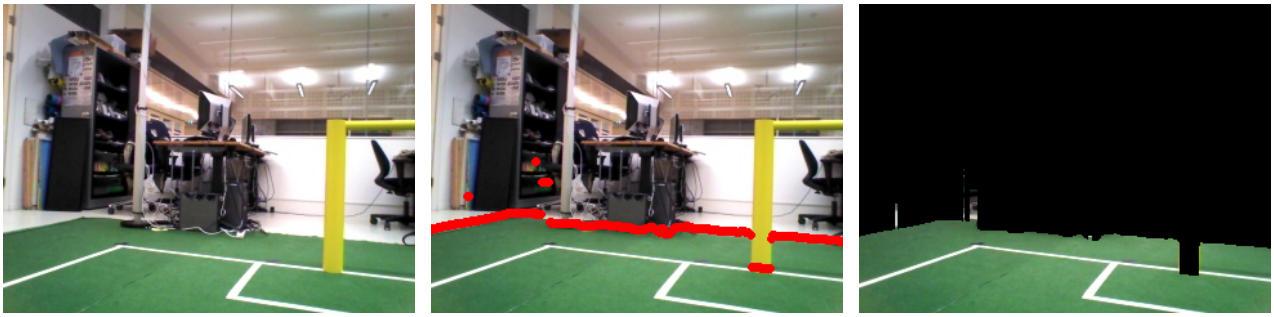
3.6.2 Our approach

As a result, we came up with our own approach of finding line segments in these noisy images. The main idea in this approach is that if we had some points on these actual line segments, we could be able to connect them based on color information, and geometric properties they have. Therefore, the first step of this algorithm, is to generate these points, scanning the picture every $5 \sim 10$ pixels vertically and horizontally. We are only interested in transitions which start from black to white to black in the end. We store the middle points as a pixel coordinates $< x, y >$ in a vector. As an illustration, we can see in figure 4 where these points are located for the same example input image as before.

Once these points are generated, the next step is to connect them in order to form line segments. For the representation of each line segment we hold a queue of points. At each time the line segment is represented by the first and the last point in the queue as starting and ending points. Pushing always the first point of the vector in the queue and deleting it from the vector the same time, we are looking for the closest 5 points to this point. From these closest points, we are checking the white ratio between these points and our initial point. Assuming that points from the same line segment do not have black pixels between them, we connect the closest one with high white ratio, usually 1.0, and pushing it in the queue of points as the last element and deleting it from the vector. The same procedure continues, but now we are looking to connect the last element on the queue with one from the remaining points in this vector. So, now the question is how are we going to stop connecting points? Considering a line formed by several points, each of these points should have a small distance from the formed line. Based on this idea, we can introduce an error function in order to define when a line starts not to comply with the points used to generate it. Assuming a queue of points Q, which contains n points, we can say that:

$$\text{Error}[Q] = \sum_{i=1}^{n} d_E\left(Q_i, \text{Line}\left(Q_0, Q_n\right)\right)^2, \qquad (1)$$

Figure 2: *Background removal process example. Left to right: original RGB image, points indicating the start of the field, regions of interest without background information.*



Algorithm 1 Line segment detection algorithm

```
1:  Input: points, image
2:  Output: line
3:  start = points[0]
4:  while points.size() ≠ 0 do
5:    line.push(start)
6:    bestCandidate = findBestCandidate(points, image)
7:    points.erase(bestCandidate)
8:    error = computeError(line, bestCandidate)
9:    if error < threshold then
10:     start = bestCandidate
11:   else
12:     storeLine(line)
13:     line.clear()
14:   end if
15: end while
```

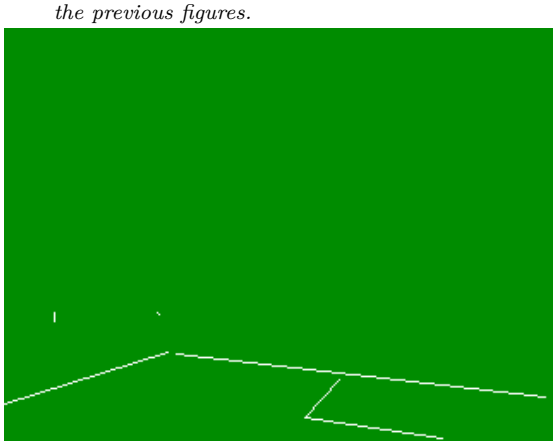
where d_E represents the shortest Euclidean distance function between two objects, and Line generates a line through two points.

This error measure proved to be really informative in our case to understand when a line is starting to consist of points which may not represent the same line segment. When this error becomes larger than a threshold, we store the produced line so far, and we continue with the next point in the vector to generate a new line. To store the line, we first check if there is an already stored line which is an extension of the current line. To find this out, we measure the error we introduced in equation 1, taking now as the line's starting and ending points the points with the largest distance between them. If the error remains small enough, and the closest points of this connection are covered in high white ratio, then we merge these lines which now represent a continuous line. Algorithm 1 presents the pseudocode for the line detection procedure. In line 12, function *storeLine* takes as input the produced line. If the line cannot be merged with an already stored line segment, it is stored as is in a vector of lines. In figure 5, we illustrate the produced lines by the above algorithm. We can see the effectiveness of our algorithm to detect lines. A big advantage over the other method we have tried is that we now detect single and continuous lines upon each actual line segment on the field.

3.7 Line Feature Detection

Until now, we have talked about image binarization and how we detect line segments. The next step in our feature extraction procedure is to identify and extract information about line features. In the RoboCup Soccer SPL field, we can distinguish 4 types of features that can be extracted out of lines. There are three lines intersections types which can be described as a T, L, and X crossing. The fourth is a circle in the middle of the field,

Figure 5: *Lines produced by the line segment detection algorithm on the above points generated by the binarized image from the previous figures.*



which crosses the middle line in two points. To detect the circle, or the ellipse as it is seen by the low-height robot's camera, we are going to talk about in a different section. In this section, we are only discussing how we can extract the first three types of intersections.

For this purpose we introduced a confidence measure which is based on geometrical properties of intersections. Given the lines which have been produced by the previous step of the process, we check every possible intersection produced by pairs of lines which have at least 10 degrees angle difference in order to avoid finding intersections by two continuous lines, and intersection is located within the margins of the image. For each one of these intersection classes, we compute a confidence measure. We can briefly define these measures for the several types:

T: the intersection is located upon the one line of the line-pair, but not close to the line's starting or ending point.

X: the intersection must be upon both lines and not close to both lines' starting or ending points.

L: the intersection must be close to both lines' starting or ending points.

These measures are computed in a way that there will always be a unique ranking among the intersection types. After the computation of the confidence measure for each intersection, we check if there are intersection types which can be combined in one. For example, two T type intersections can form a X type if they are located near and have opposite orientation. Except from the computation of the intersection position and confidence for each type of intersection, we also compute the orientation of each of them, in order to deliver as much information as possible to the localization system about the robot's position.

3.8 Goal Detection

Last step in the feature extraction process is the detection of the goal. Goal detection is critical for the robot in order to take sensible actions, and due to the fact that it is the only easily distinguishable landmark in the field. Both goals on the RoboCup Soccer SPL field have a distinct yellow color. We have already discussed how the binary image in respect to the goals is created. Integrated in the same scan of the input image, we compute two different histograms for yellow pixels, one horizontal and one vertical. These histograms are going to help us later in goal detection. We also store points of yellow-green transitions during the same image scan to estimate the possible goalposts' bottoms.

Goals consist of two vertical bars and a horizontal attached to the tops of the two vertical bars. So, we can realize that vertical bars are more informative, especially to inform our localization system about the orientation in respect to the goalposts.

The first step in the goal detection process is the use of the horizontal histogram of yellow pixels in combination with the y-positions in where we found yellow-green transitions. Combining this information, we can find the local maxima on the horizontal axis and find the positions where goalposts are located. If the maxima of this histogram have values under a certain threshold, the process will not continue for vertical posts detection, as it appears that there is no vertical post. If these values are high enough, and we have two local maxima, we are pretty sure that there are two vertical posts in the image, if there is only one we are looking only for one vertical post. In both cases we scan the image to left and right from the maxima and exclude parts of the image which do not contain yellow pixels at all, for computational efficiency.

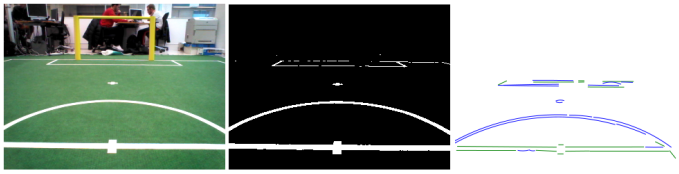
The next step is the line detection, using the same approach we have discussed earlier, but now generating points only with a vertical scan, as we are only interested in vertical lines. From these vertical lines we choose the best ones with respect to their length, orientation, and with the prior knowledge about possible horizontal positions for the goalposts. These lines will be extended towards the bottom of the goalpost and the top, and by doing so estimating the true height of each goalpost. To estimate the width, we are taking random samples upon the line representing the goalpost and measure with a simple scan the width. The average of all these measurements will be our estimation about its width. We also compute a confidence measure about the bottom of the goalposts, taking the estimated bottom of the representative line and computing the distance from the yellow-green transition discussed earlier. The same procedure is applied for the horizontal goalpost as well. Horizontal goalpost can be very informative in cases that only one vertical goalpost is visible by our agent. The orientation of the horizontal goalpost can be used in order our agent to distinguish between left and right goalpost. Figure 6 illustrates the results of the goalpost detection in three different situations.

3.9 Ellipse Detection

The field of the RoboCup Soccer SPL league has a circle in the middle with a diameter of 120 cm. This is an important feature for the NAO's self-localization as this is the only unique feature on the field. In computer vision, most commonly ellipse detection is carried out on binary images from an edge detector. However, in general, ellipse detection does not address the problem of false positives and false negatives well, especially for partially visible ellipses.

For this project, we chose to implement an ellipse detector based on Pătrăucean's method [**ED**]. We chose this technique because it is parameterless and required no special tunining for

Figure 7: *Left to right: RGB image obtained from NAO's lower camera, binary image of field lines after background removal, possible ellipses detected (and lines, not considered)*



different scenarios. Most ellipse detectors require as input certain parameters that define the scenery such as a range of expected radii of ellipses, and a minimum distance between two ellipses. This approach aims to be free of critical parameters as much as possible, in order to avoid introducing false negatives. Also, it provides a control over the number of false positives by using *a contrario* validation. The greedy-like candidate selection step implemented by this algorithm ensures a reasonable execution time. The two primary steps in this algorithm are:

- Region growing: Pixels with a similar gradient orientation are aligned to form rectangular regions.
- Region chaining: These detected regions are chained together if they roughly follow some conditions that define ellipses.

To tackle the problem of false positives, *a contrario* validation is employed to reject or validate candidate ellipses. The *a contrario* theory employs the non-accidentalness principle (which says that "we see nothing in noise"). Therefore, as described in [**ED**], candidate ellipses that have a higher chance to be observed in noise are discarded. However, as can be seen in figure 7, even after this step, we still find a number of false positives because small features like penalty X crossings, and resedue background noise often gets classified as small arcs by the ellipse detector. This problem can easily be tackled by filtering the candidate ellipses on their arc lengths and putting a minimum threshold on it.

The ellipse detection step can also ensure fewer false positives (and faster computation) during the line feature detections stage if ellipse detection is performed first: if we know with confidence the area that the circumference of the ellipse is covering, we know that line crossings do not exist on it and hence we can mask this area for the line feature detection step.

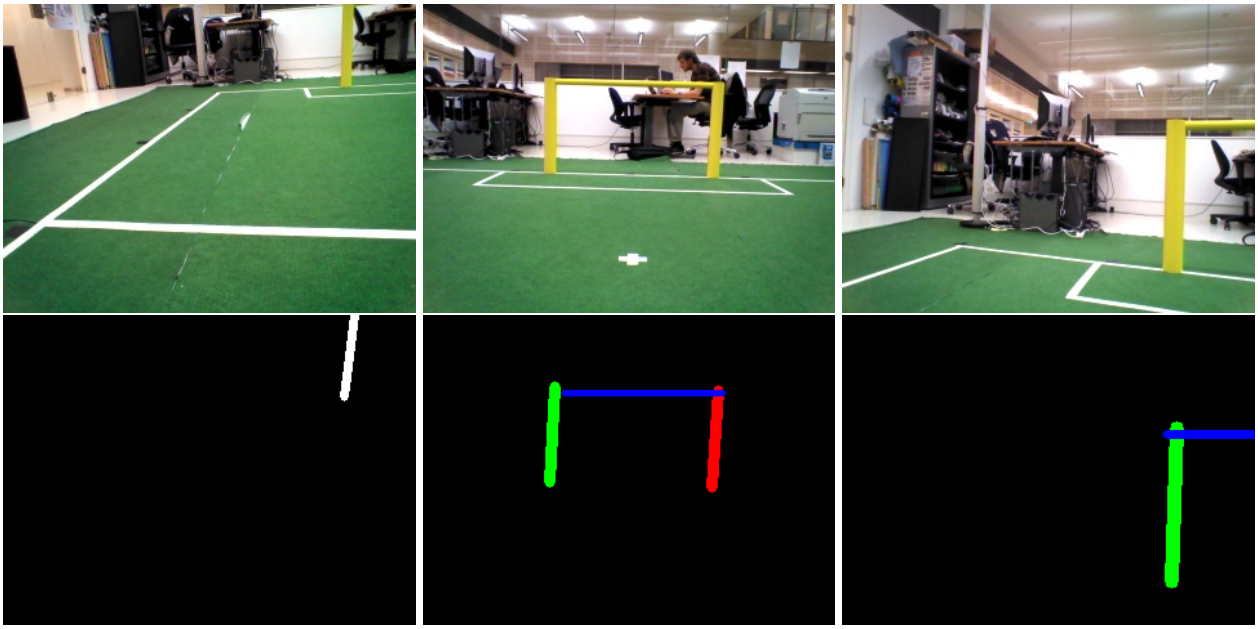
3.10 Inverse Perspective Mapping

A transformation from one projective plane to another can be utilized for generating views of objects different from the camera position. Inverse perspective mapping (IPM) makes use of this. It projects an image on a known plane and gets rid of the perspective effect on objects that are on that plane and produces a distorted view for all others [**IPM2**]. We have not found other teams using this method.

Information like the angle of view (under the horizon) and the distance of objects from camera associate a different information content with each pixel of an image. In fact, the perspective effect must be considered during the processing of images so as to weigh each pixel with respect to its information content. As a counter to this issue, IPM lets us remove the perspective effect and remap the image to a 2D domain, so that the information content is evenly spreadout among all pixels [**IPMWeb**].

It must be noted that applying the IPM transform requires some camera conditions to be known. These are the camera's height above ground level, its aperture, its tilt with respect to the vertical axis and its resolution. Since this information

Figure 6: *Goal detection example results (green: left goalpost, red: right goalpost, white: undefined vertical goalpost, blue: horizontal goalpost.*



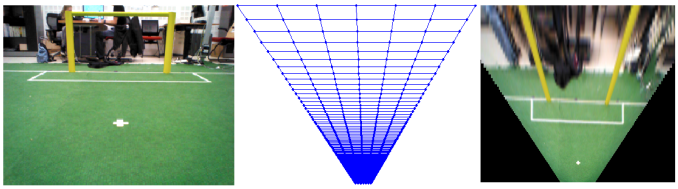
can be computed fairly accurately by the NAO through inverse kinemetics, the IPM transformation can be used by it. Thus, assuming that the field in front of the camera is planar, a bird's eye view of the playing field can be obtained.

In the domain of robot soccer, the use of IPM transformation can have a few potential advantages. Estimation of distance and bearing to a feature point on the field is simplified because now we have a bird's eye view of the field in front of us. Preprocessing steps like background removal is also much easier than on regular camera images because of the fact that always the lower part of the image contains the field. Moreover, IPM also overcomes the problem that the NAO's head can tilt along various axes because this transformation is robust to such viewpoint changes. Lastly, the feature extraction step can itself be made more robust and computationally inexpensive. This is because when viewing the field from a top perspective, all intersecting lines on the field appear to intersect orthogonally and circles retain their shapes. However, one of the main problems we faced while using IPM for feature extraction was that feature points far from the camera appear to get distorted due to the transformation. This causes many false negatives while extracting features. One ad-hoc solution to this problem was transforming the binary image of the lines in the field instead of the raw RGB image. Another problem with using IPM was the difficulty in measuring projected objects such as goal posts on the field. The transformation always stretches the goal posts such that their top portions are no longer visible/detectable. Thus, goal post detected continues to rely on the original camera images.

We now briefly explain this transformation[**IPM**] while not going into the details of its mathematics. In IPM, we trace a ray from a point in the image plane through the center of projection and towards the horizontal plane, where it results in an intersection. In this way, we find the corresponding image point.

We first create matrices that contain the x and y locations in the world frame, where the pixels in the bottom portion of the camera image (where we assume we see the field based on the camera's viewing angle, height and aperture) are mapped by using the inverse perspective transformation equations (see figure 8, center). Next, we find the weights needed to create an inverse-perspective-mapped image with evenly spaced pixels

Figure 8: *Left to right: front facing RGB image obtained from NAO's lower camera, map of image pixels to the world frame, bird's eye view obtained after IPM*



by linearly interpolating between intensity and color values in the original image. Finally, we apply the inverse-perspective interpolation mapping to an RGB camera image to get an inverse-perspective-mapped image[**IPMCode**] (see figure 8, right).

4 Localization

We based our localization module on the Monte Carlo Localization method (MCL). The module has been augmented with several extensions taken from literature. This part of the report describes the standard MCL approach, the models that were used to incorporate incoming sensor information (i.e. odometry and observed visual features in the camera image). We describe how the observed features are associated with the given feature map of the soccer field. Eventually we elaborate on how the standard approach was improved to overcome well known shortcomings of the usage of particle filters in a robot soccer tournament environment.

We chose to use the MCL as basis for localization of the robot since it is a well known approach, and thoroughly researched. It is a multimodal particle based localization method (i.e. it can deal with multiple pose hypothesis at the same time). Since the new environment for 2013 is completely symmetric this is a valuable property. In addition to that, MCL can deal with erroneous sensor information, which is an unavoidable problem in real life applications in robotics. It is able to recover from kidnapping (i.e. repositioning the robot on the field without odometry information). Finally the number of particles can be adjusted. This is useful, since the NAO robot platform offers

only limited computational resources.

4.1 Monte Carlo Localization

The basic MCL algorithm is depicted as pseudocode as described by Fox et al. [**MonteCarloLocalization**] in algorithm 2.

Algorithm 2 Monte Carlo Localization

1: **Input:** χ_{t-1}, u_t, z_t, m
2: **Output:** χ_t
3: **monte_carlo_localization**$(\chi_{t-1}, u_t, z_t, m)$
4: $\bar{\chi}_t = \chi_t = \emptyset$
5: **for** m=1 to M **do**
6: $x_t^{[m]}$ = sample_motion_model(u_t , $x_{t-1}^{[m]}$)
7: $w_t^{[m]}$ = measurement_model(z_t, $x_t^{[m]}$, m)
8: $\bar{\chi} = \bar{\chi} + \langle x_t^{[m]}, w_t^{[m]} \rangle$
9: **end for**
10: **for** m=1 to M **do**
11: draw i with probability α $w_t^{[i]}$
12: add $x_t^{[i]}$ to χ_t
13: **end for**

where m denotes the map, χ_t particles at time t, u_t control input (in our case odometry information) at time t, z_t sensor input (in our case visual features) at time t.
In general the MCL consists of three main steps:
Initialize: place particles at random

1. **process odometry information:** Change particle poses (position and rotation) accordingly. (line : 6)
2. **process visual sensor input (extracted features):** Calculate likelihood of pose hypothesis (particle) and weight particle accordingly. (line : 7)
3. **resample:** Redraw all particles, place new particles distributed according to the former computed weights, and reset all weights to one. (for loop starting in line : 10)

The initialization of particle poses does not necessarily have to be at random, if the initial robot's position is known, or approximately known (e.g. only one side of the field) it is advantageous to sample from this known prior position.

Everytime new information is available those three steps can be executed, first process the odometry information, and move all particles accordingly. Since the odometry of robots in the majority of cases incorporates an error, we add additional (Gaussian) noise to it.

Then we process the visual features, which in our case will be returned from the feature recognition module. Each feature comes with range, bearing and type. In order to compute a likelihood for each pose hypothesis, we need to associate each given feature to a feature of the feature map. This process is described in detail in the following section. Finally we resample all particles according to the likelihood of each pose hypothesis. Such that at poses that had a high likelihood and thus a high weight, we place more particles. In this step all weights will be reset to one. And we repeat all steps with new incoming information.

4.2 Sensor Model

The feature recognition module returns the range and bearing and type of a feature extracted from the image feed of the NAO robot (an elaborate description of the methods used can be found in section 3). Currently L-crossings, T-crossings and X-crossings and the bottom of the goal posts can be processed (it is possible to incorporate more types of features, e.g. lines, or the middle circle). These features occur multiple times in the map, it is not possible to distinguish between different features of the same type a priori. For the Monte Carlo Method to work, we need to associate one feature to a feature on our feature map of the soccer field. To determine which observed feature will be associated with which map feature, we compute the observed feature position according to our current pose hypothesis to map space (assuming that we are positioned in the particle position, whose weight we want to compute), then choose to associate it with the feature in the feature map which is closest:

$$\underset{j}{\operatorname{argmin}}(\sqrt{(m_{j,x} - f_x)^2 + (m_{j,y} - f_y)^2})$$

where f is the feature observed, f_x and f_y the feature position and $m_{j,x}$ and $m_{j,y}$ the position of feature j in the feature map with the same type as observed feature f.

(This simple model was chosen due to time constraints. More elaborate models are possible, e.g. aggregate two or more features and compute correspondence values for those "feature patterns"). To compute the final likelihood, we calculate the range and bearing of the feature map feature chosen, according to our pose hypothesis. Again we assume erroneous sensory information. For simplicity we model this error as a Gaussian, although the underlying error model could be different. The sensory model computes the final likelihood of a pose estimate as described by Thrun et al. [**ProbabilisticRobotics**]:

$$q = \prod_{i=1}^{N} prob(r^i - \hat{r}^j, \sigma_r) * prob(\phi^i - \hat{\phi}^j, \sigma_\phi) * prob(s^i - \hat{s}^j, \sigma_s)$$

where q is our final likelihood estimate for a certain pose hypothesis, i = 1 .. N the number of features observed in current image of the camera, r^i the range of the observed feature i, $\hat{r}^j$ the computed range of the feature j in the feature map associated with the observed feature according to the current pose hypothesis, $prob(r^i - \hat{r}^j, \sigma_r)$ denotes the probability of value $r^i - \hat{r}^j$ in a zero mean Gaussian with variance σ_r. $prob(\phi^i - \hat{\phi}^j, \sigma_\phi)$ and $prob(s^i - \hat{s}^j, \sigma_s)$ denote the probability of a zero mean Gaussians for difference in bearing of observed feature ϕ^i and bearing of associated map feature ϕ^j and probability of $s^i - \hat{s}^j$ on a zero mean Gaussian with variance σ_s for the difference in certainty of having chosen the right association of feature on the map and observed feature accordingly.

4.3 Odometry Model

The odometry model can be either implemented from actual sensor data (through built in accelerometer, or computation of steps taken), or the control input. Since the current implementation was not tested on the NAO robot platform yet, it is still left to further evaluation which input is more advantageous. Since both methods incorporate an error, control input through not exactly working motors, or sensor information noise, for each particle we sample from the odometry input with additional noise. This noise is assumed to be Gaussian and independent. At the moment the odometry noise is modelled as error in x and y direction travelled and noise for additional rotation.

4.4 Augmented Monte Carlo Localization

Since the standard MCL algorithm described in section 4.1 might perform poorly in case of kidnapping. This is if the confidence of the robot pose is high and it gets replaced from an extrinsic source (e.g. penalty of referee). If the pose confidence is high, the particle density is high around the most probable point in pose space. The robot might get replaced to a pose where no particle is located. It might take a long time till particles

will aggregate at the new position. A suitable extension to the standard approach is to keep track of the measurement likelihood both in a short term average and a long term average. If those differ, the robot is capable of detecting a kidnapping situation. New particles can be sampled at random according to the difference of short and long term average. Thus increasing the recover speed after a kidnapping situation as described by algorithm 3.

Algorithm 3 Augmented Monte Carlo Localization as depicted by Thrun et al. [**ProbabilisticRobotics**]

1: **Input:** χ_{t-1}, u_t, z_t, m
2: **Output:** χ_t
3: **augmented_monte_carlo_localization**$(\chi_{t-1}, u_t, z_t, m)$
4: static w_{slow},w_{fast}
5: $\bar{\chi}_t = \chi_t = \emptyset$
6: **for** m=1 to M **do**
7: $x_t^{[m]}$ = sample_motion_model(u_t , $x_{t-1}^{[m]}$)
8: $w_t^{[m]}$ = measurement_model(z_t, $x_t^{[m]}$, m)
9: $\bar{\chi}_t = \bar{\chi}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
10: $w_{avg} = w_{avg} + \frac{1}{M} w_t^{[m]}$
11: **end for**
12: $w_{slow} = w_{slow} + \alpha_{slow}(w_{avg} - w_{slow})$
13: $w_{fast} = w_{fast} + \alpha_{fast}(w_{avg} - w_{fast})$
14: **for** m=1 to M **do**
15: **with** probability max{0.0, 1.0 - w_{fast}/w_{slow}} **do**
16: add random pose to χ_t
17: **else**
18: draw $i \in 1, .., N$ with probability α $w_t^{[i]}$
19: add $x_t^{[i]}$ to χ_t
20: **end with**
21: **end for**

To incorporate the long and short term average we change the basic algorithm described in section 4.1 as follows:
We define the long and short term average in line 4, insert the computation of the current weight average in line 10, then update w_{slow} and w_{fast} in lines 12 and 13. With α_{fast} and α_{slow} being the decay rates, where $0 <= \alpha_{slow} << \alpha_{fast}$. Finally we sample particles at random with probability $max\{0.0, 1.0 - w_{fast}/w_{slow}\}$ and otherwise according to the particle weights computed in the measurement step, as before. This is called Augmented Monte Carlo Localization (AMCL).

4.5 Resetting Sensor Model

Now the robot is to a certain extend able to detect a kidnapping situation (this might take several iterations of the AMCL till the long and short term average differ sufficiently), and injects particles at random. In the case of injecting particles totally at random, the localization module is subject to pure chance to sample particles near to the actual position. We can speed up the process in case we have observed visual features. E.g. if we see both goal posts, it is highly unlikely that we are in a position from where we actually cant see any goal. In this case it is possible to sample from the visual sensor information instead of totally at random a similar method was originally proposed by Lenser and Veloso [**lenser2000sensor**]. In our implementation we sample from one feature at random with added Gaussian noise in range and bearing. Since we have several features of the same type in the feature map, again we choose one at random for each sample we want to resample. This will result in several circles around all features on the feature map of the type of chosen visual feature to sample from. Due to time constraints, we only chose one observed feature to sample from. It might increase the recovering speed after a case of kidnapping, to sample from the pose probability distribution of several features (we can reduce the sample space from several circles around the features in the feature map to several points, if we observe and take into account two visible features).

4.6 Resampling method

As described by Thrun et al. [**ProbabilisticRobotics**] resampling using a random number leads to loss of variance. (e.g. a robot with no perception, after a sufficient amount of resampling steps would only remain with one position hypothesis. Since if sampling with a random number for each particle might lead to the loss of one hypothesis, if two particles are resampled at the same pose). Algorithm 4 depicts low variance resampling.

Algorithm 4 Low variance resampling

1: **Input:** χ_t, W_t
2: **Output:** $\bar{\chi}_t$
3: **low_variance_resampling**(χ_t, W_t)
4: $\bar{\chi}_t = \emptyset$
5: r = rand(0;M^{-1})
6: c = $w_t^{[1]}$
7: i=1
8: **for** m=1 to M **do**
9: U = r + (m-1)* M^{-1}
10: **while** U > c **do**
11: i = i + 1
12: c = c + $w_t^{[i]}$
13: **end while**
14: add $x_t^{[i]}$ to $\bar{\chi}_t$
15: **end for**

Where X_t particles with weights W_t at time t, and M the number of particles. This method creates only one random number, and adds a fixed step size iteratively to a "pointer" (U). In addition to that we keep track of the sum of particle weights such that, for every step we sample one particle. The particle will be sampled with pose values of particle which "weight bin" (c) the "pointer" is currently pointing. This method ensures the systematical coverage of the sample space, and no samples are lost. We additionally changed the algorithm slightly, instead of drawing a random number between 0 and M^{-1} (line 5) we compute the average particle weight beforehand. This also applies for adding step size in line 9, we add the average particle weight instead of M^{-1}. This is done to evade complications in case the assignment of particle weights in previous steps is not totally statistically sound.

4.7 Visualization

For debugging purposes we developed a simple visualizing modules built on basis of OpenCV[2]. This module reads the parameters of the football field from a parameter file, and can draw the particles, the pose hypothesis of the AMCL. It proved to be very useful, and might be used for future projects.[3]

5 Evaluation

We have evaluated the line crossing detection module and goal post detection module on a small dataset of 10 images, which should be representative given our description. The 10 images contained 17 L crossings, 15 T crossings, 8 X crossings, and 17

[2] `http://opencv.org/`
[3] The code can be accessed at: `https://github.com/pkok/Robolab/tree/master/NaoLeague/LocationVisualizer`

goal post parts. The first module has a precision of 0.90 and a recall of 0.65. The goal post detector has a precision of 1.00 and a recall of 1.00. These high scores for the goal post detector are most probably due to the fact that there are no other yellow objects in the image which are considered foreground.

6 Discussion

Due to time constraints, the described and implemented parts of the system were only compiled and tested on standard computer hard- and software. The code still has to be tested on the limited resources of the NAO robot.

6.1 Feature detection

All tests of the feature recognition were conducted on still images, taken from the NAO's bottom camera. The images of a moving robot will contain a considerable amount of blur, which might require additional tricks to make the whole system robust enough in an actual soccer game, e.g., only classify detected features if they can be observed in a sufficient number of frames.

The existence of other robots on the field will increase the number of false positives the feature recognition will detect.

While the feature extraction works very fast on standard computer hardware, it might be desirable to optimize these routines. For example, one could use the output of the localization module for searching for features at more specific areas of the camera image. This more educated approach could speed up the feature extraction, as this can be used to eliminate the currently used brute force for checking every pixel. This might be realized by using the optical flow in the image, or implementing a different tracking method.

6.2 Localization

The Monte Carlo localization has a number of parameters, which influence the performance to a great extend. Although the algorithm can be adjusted to less processing power platforms by decreasing the number of particles, there might be some space to decrease the computational costs by tweaking the code. Besides that, both the feature recognition and localization do not have to be executed for every new frame. It might be beneficial to postpone computation of features and location updates if the robot does not move, or the computational resources have to be used for other computations.

The localization can be improved performance-wise as well. We now have implemented an augmented Monte Carlo localization algorithm, which can easily be confused about on which of the two symmetrical halves you are. For example, B-Human [**TeamReportB-Human**] models only half of the field explicitly. The other half is modeled by using filters that keeps track of the likelihood $\mathcal{L}$ that this robot is on a certain field half. By implicit modeling of the second half of the symmetrical field, each particle is twice represented with likelihood $\mathcal{L}$ to be on one side of the field, and $1 - \mathcal{L}$ to be on the other half. On the other hand, rUNSWift does not use a particle filter at all; since 2011 they have been "experimenting with multi-robot-multi-modal Kalman filtering and researching the application of loopy belief revision to achieve similar benefits, but more efficiently, with larger teams" [**rUNSWift-TD-11**]. Several teams are using self-detected visual features as another source of information for the localization algorithm [**sturm2006msc**, **Cuauhpipiltin-TD12**].

Another method one could experiment with is triangulation by audio signals. When at least two NAOs are quite sure about their position, the first could send out an audio signal, which robots unsure about their position will listen to. The sender will also announce its estimated location over WiFi. Other robots which are certain about their position can do something similar, so that the uncertain robot can estimate its position through triangulation from the bearings. However, this does rely heavily upon the precision with which the NAO can detect te bearing of the sound.

7 Conclusion

This report proposes an implementation for the Dutch Nao Team's localization module. While the code has not yet been compiled for the NAO platform, it has proven to run quite fast on standard computer hardware.

We have implemented a feature extraction module which heavily depends on a line detection algorithm which is designed by us for this project. A first inspection shows this algorithm to work well. We have implemented ellipse detection, but these features have not yet been integrated in the localization module. The distance from the robot to a feature is computed with an inverse perspective mapping, which, in future work, can be used for easier feature detection. The localization module is based upon the augmented Monte Carlo localization algorithm.

For both modules we have suggested possible improvements in 6, besides more thorough testing.