Humanoid Robot Obstacle Recognition via Data Filtering, Localisation and Robot-to-Robot Communication in Context of RoboCup

Benjamin Scholz, Daniel Speck, and Judith Hartfill

University of Hamburg, Fachbereich Informatik
2scholz@informatik.uni-hamburg.de,2speck@informatik.uni-hamburg.de,
2hartfil@informatik.uni-hamburg.de

Summary. The abstract should summarize the contents of the paper. It will be set in 9-point font size and be inset 1.0 cm from the right and left margins. There will be two blank lines before and after the Abstract. Use the abstract section to provide a teaser for the contents of your report I Do not attempt to write a review or summary I Be concise: Your abstract should have 200 words or less (do not use more than 250 words)

Key words: obstacle recognition, obstacle avoidance, localisation, filtering, data smoothing, vision, swarm intelligence

1 Introduction

In this project our aim was to implement obstacle recognition for a team of robots taking part in the RoboCup Soccer Humanoid Kid Size League. So far each robot in the team made decisions due to its own information only such as the field of vision. This led to a decision making being dependent on one robot's limited individual data.
Our objective was to develop modules which share data not only on one robot but between all team mates for realizing an intelligent behaviour via localisation and mapping obstacles in this world model.
For implementing such modules it is required to have sufficient vision data of the environment. Gathering this information we noticed that the supplied quality does not match this task's demands.
Furthermore the position data of the robot and other objects were noisy, so data filtering was necessary.
Intelligent behaviour requires a world model which provides its information in a transformable and unique way between modules or even robots. Fulfilment of this task requires an internal team communication system contributing one robot's information for every team member.
Thus a form of swarm intelligence could increase the quality of the robots acting together as a whole team by increasing the quality of information.
Playing football always requires communication, so sharing information improves

the team's efficiency. Transfering this key aspect of football to the software was one of our main ambitions.

1.1 Comments on group work & split of labour

1.2 Motivation

Obstacle recognition and dynamic behaviour, specifically pathfinding towards or around obstacles is important if several robots shall act together as a team in a certain way to achieve specified goals.
Team play/efficiency, fair play and hardware costs are the main reasons for intelligent obstacle recognition. One aspect of fair play is to avoid the physical contact between robots and as long as their hardware sometimes loses balance when the robot is touched, it is very likely to fall and - in worst-case-scenario - causes other robots downfall. Such collisions usually cause high costs for hardware fixes and also disturbs the gameplay. Getting up again takes several seconds and in addition it is not within the meaning of fairplay. Furthermore physical contacts are not compliant to the rules.
Moreover and especially better knowledge about the surroundings is strategically important in the RoboCup competition. Having a reliable prospect of its own position, the position of the ball and the opponents' position combined with adequate path finding algorithm can raise the number of scores significantly.
Those aspects cause a big increase in making the robot football play more realistic and authentic, which is the main aim of the RoboCup competitions.

1.3 Problems

Vision For an intelligent obstacle recognition several sufficiently pre-processed data input is needed to localise the obstacle itself, localise the robot itself, classify the obstacle (ball, goal, teammate, enemy, other obstacle) and develop an adequate strategy of (re-)action.
One of our first problems were the already implemented algorithm for obstacle detection and goal detection. We discovered that neither obstacles nor goals were tracked/recognized satisfactorily for our tasks.
The obstacle detection was too imprecisely to fulfil our plans of an intelligent reaction and the biggest problem was the missing localisation of the robot itself which is realized by tracking the goals (via tracking the two goal posts). Even when the robot stands still completely in a distance of two metres directly in front of the goal the vision detects four to six goal posts most of the time.
The majority of false positives were shadows of the real posts or random mirages/reflections on the walls. This resulted in "jumping" goal posts because for the behaviour two posts are selected and out of the four to six possible posts the selection process to distinguish between the possibilities was not stable.
In addition to that even when the real goal posts could be identified for several frames the algorithm often was not able to calculate the correct relative distances to the goal posts.

The algorithm was designed to draw a rectangle over a possible goal post but it has no specific filters for the dimensions. This leads to rectangles having a wrong position, height and width compared to the original post. Some recorded datasets showed rectangles with twice or half the size of the real post and positional failures of several 10 centimetres.
Appropriately reacting behaviour systems e.g. for intelligent obstacle recognition, localisation and further path planning or even postprocessing filters would fail on such raw data.

Filtering The whole topic of obstacle recognition relies on information which is extracted out of (heavily) noisy input data.
For our solutions primarily the goal post data and the later calculated goal centre information is important to calculate real, absolute coordinates for localisation purposes on the field out of the relative camera input.
The raw data of goal posts (relative tuples of x, y coordinates) when recorded over several seconds (with a still standing robot) oftenly showed a spreaded distribution of points in a circle with a radius of nearly half a metre. Those datasets of 100 to 200 recorded samples had a standard deviation of 200 millimetres and higher.
Having such noisy data renders absolute positioning tasks and intelligent obstacle recognition (avoidance in case of real obstacles, path planning in case of balls) impossible.
Since the robots do not have auto-calibration features the calibration of the camera and motor angles deteriorate the noise problem. Once the calibration is manually done its configuration is static but the real values vary other time, for example when the robot falls accidently.
In addition to that the manual calibration is expansive in terms of time.

Localisation Detecting an obstacle isn't worth much without knowing our own robot's position on the field. Of course the robot could avoid the recognised obstacle, but there is no way of knowing if it benefits our team in this situation. If the robot bypasses an opponent but actually walks in the wrong direction we haven't gained anything.
Also there is no way of knowing what robots are our team members and which one are our opponents.
If the robot had all this data it could make more intelligent decisions and improve the flow of play.
To achieve these goals we need a world model we can store information in, like our own position and the position of our team members and opponents.

Communication When thinking about how to achieve most and best information for the robot, we thought about swarm intelligence mechanisms. Althought our Darwin-OP platfors do have a lot more computing capacity then usual swarm intelligent bots have, we still can benefit from researches in this part of robotics. particularly as communicating is a very important part in "real" human soccer.

2 Solutions

2.1 Hinweise (deleten!)

What did you do and how did you do it?
Methods
Design
Implementation
Do not include every possible detail and avoid redundancy
Use subsections to emphasize certain aspects/components of your work - but do not overuse them!
Avoid the passive voice: Y was done by X, use the active voice: X did Y

2.2 Vision

The visions detection of obstacles and objects is based on a horizon scanning algorithm which tries to find the horizon in the current frame. Basically it reads the pixel information of the current frame in matrix and finds the specific row where the horizon is located. Once this row is found the algorithm for obstacle detection scans down to find "green" pixels for locating the field. Every obstacle, for example the goal posts, create non-green shapes in this matrix.

Goal information The algorithm searches for the yellow color mask of the goal posts to detect every goal post in the current frame. For every potential goal post which is found the algorithm tries to extrapolate the shape of the goal post and to fit a virtual rectangle with nearly the same dimensions on it.
Often the width of the rectangle was calculated wrong (beeing twice or half as big as the original post). This failure exists because of image distortions which manipulate the objects shapes and the angle under which the object is seen (basically the bottom of the object is wider, the top thinner because of the angle). This problem basically remains but with modified parameters we were able to make the fitting of the virtual rectangle more strict. Those changes slightly improved the rectangle width compared to the real goal post.
In addition to that the implemented algorithm now searches the bottom of the object by scanning down to one of the objects last lines in the matrix. This solution relatively stabilizes the problem because the previous algorithm just picked some line of the top of the object compared to the horizon line in the current frame and those information varies much more (top & horizon line) in dependency of the camera angle compared to the objects bottom.
Scanning for the bottom of the object is even more profitable when it comes to the height of the object because the bottom of a goal post (yellow) and the field (green or white) always have a hight contrast change so the bottom line is found reliably.
To further improve the detection of the goal posts we finally modified the algorithm not to take a corner of the virtual rectangle but to calculate the center of the bottom line. This approach is more stable because the error is halfed when the width of the virtual rectangle changes.

Obstacle information The obstacle detection also scans downwards starting at the horizon line in the current frames matrix. Abnormal contrast changes are interpreted as obstacles if its size is large enough.
Objects like the ball are filtered by another algorithm. A point cloud filtering and an orange colour map are detecting the ball.
Arbitrary coloured obstacles with a significant size are classified as miscellaneous obstacles. Those obstacles are "real" obstacles which should be avoided (in case of a human on the field or an enemy robot) or blocked (in case of an enemy with the ball trying to score at our goal).
Those obstacles information is processed separately and can be filtered to stabilize its data / position. Those data is later mapped into our grid world model.

2.3 Filtering

Data noise The majority of the bitbots code works without filters on raw data, especially in the case of goal information no postprocessing exists.
The goal information (goal post positions and goal center position) is needed to localise the robot on the field in our case. Intelligent behaviour according to obstacle recognition relies on this localisation (our grid world module) so the goal information is the most important data for our software and has to be as accurate as possible.

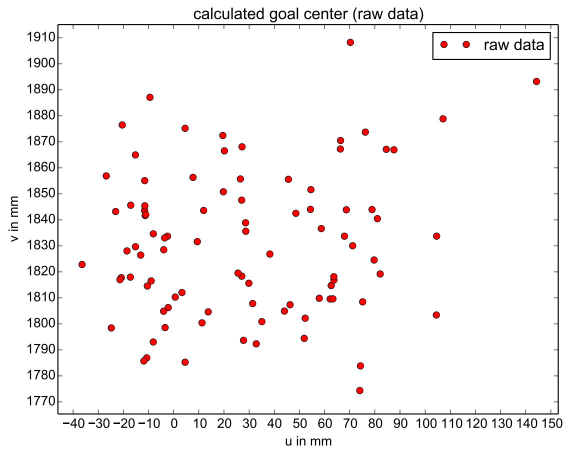

Fig. 1: Plot of raw goal center data (100 data sets recorded in about 5 seconds) without filtering and calibration. Original vertical distance to goal center was 2000mm and horizontal distance 0mm. The robot was standing still.

As shown by figure 1 the noise of the raw data in combination with missing calibration is too high to allow an accurate localisation.

We define a distance vector $\mathbf{d} := \begin{pmatrix} u & v \end{pmatrix}^T$ to contain the horizontal (u) and vertical (v) distance to a point relatively to the robot's position.
The vector $\mathbf{d_0} := \begin{pmatrix} 2000 & 0 \end{pmatrix}^T$ is the real vector to the goal center and vectors like $\mathbf{d_1} := \begin{pmatrix} 1797 & -25 \end{pmatrix}^T$ and $\mathbf{d_2} := \begin{pmatrix} 1877 & 107 \end{pmatrix}^T$ are the vectors of two recorded points shown in figure 1.
$\mathbf{d}_0$ differs to $\mathbf{d}_1$ by $204 = \sqrt{(2000-1797)^2 + (0-(-25))^2}$ and $\mathbf{d}_0$ to $\mathbf{d}_2$ by $163 = \sqrt{(2000-1877)^2 + (0-107)^2}$ in euclidean distance.
Those differences are too large to ensure a correct mapping in our grid world module as the differences occurred in less than three seconds while the robot was standing still.
Nevertheless clusters of points can be detected in figure 1 and this is the reason why we decided to filter the data via a DBSCAN algorithm.

DBSCAN Our approach to reduce the noise on data was implementing a two dimensional version of the DBSCAN (density-based scanning) algorithm [1], which we slightly customized to our needs.
At first our code was an exact re-implementation in pure Python of the DBSCAN algorithm [1] originally proposed by Martin Ester et al. in 1996 but this procedure was just for testing to assure the algorithm meets our requirements.
Due to the performance issues of pure Python implementations we refactored the complete code and made use of Numpy (mainly) and other SciPy libraries. As a consequence of the C/C++ core implementations of those libraries the overall execution time of our code decreased.
In addition to that the refactored, performant version of our DBSCAN is implemented object oriented. This leads to an overhead in space complexity but offered easier debugging as well as an additional improvement of execution time because the calculations are saved in objects and those cached results are re-used as much as possible.
In our tested cases the pure Python implementation ran about 8 to 12 times slower on average (dependent on current background tasks) on our darwin-op hardware compared to our current object oriented Python implementation with Numpy and other SciPy libraries.
Those optimizations are essential considering the very limited computing power of our current darwin-op robots.
For calculating the goal information each goal posts raw input data (if post is seen) is recorded separately in buffers of a maximum size of 60 items. Every new frame delivers one new data set and on average the robots architecture is able to handle about 20 frames per second. If a buffer is full, the oldest entry is dropped and the newest saved. As data structure for the buffers we chose Python Deque objects for easy appending and extracting data sets.
If the robot changes its position, old data sets would mess up the calculation, so our DBSCAN object automatically drops data sets which are older than 10 seconds to prevent those kind of failures. Minor positional changes in the meantime are no problems in most cases, because the DBSCAN would filter those datasets

appropriately.
In figure 2 a graphical illustration of the DBSCAN algorithm is shown.

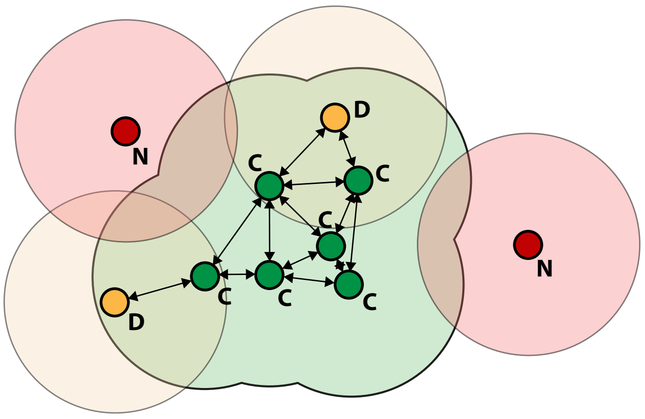

Fig. 2: DBSCAN graphical illustration. The points labeled with 'C' are core points, the yellow ones labeled with 'D' are density-reachable points and the red ones labeled with 'N' noise points. The minimum points parameter is set to 4 in this example.

The DBSCAN algorithm has two parameters: the minimum points κ (in some papers referred only with "minimum points"), which are required to initialize a new cluster and the radius ϵ in which neighbouring points should be searched. The point counts itself to its neighbourhood, too. Therefore every point has a κ of at least 1.
All points are two-dimensional and consist of the relative horizontal distance (u) and the relative vertical distance (v) to the object.
At first DBSCAN marks every point as "unvisited". In the next step the algorithm chooses one point p_0 at random, marks it as "visited" and searches in his neighbourhood for neighbouring points within its ϵ radius. This is done by calculating the euclidean distance between the points.
This procedure would find all points within a circle with the radius of ϵ as shown in figure 2. If the amount of found neighbouring points in neighbourhood to p_0 is greater or equal to κ, than a new cluster is initialized. Otherwise the point is would be marked as "noise".
A newly initialized cluster would then continue with the next point p_1 within the cluster and also mark it as visited as well as search for other neighbouring points in p_1's ϵ neighbourhood. This continuation stops when all points in the cluster are processed, so the cluster is expanded to its maximum size like the green cluster in figure 2.
Points within the cluster's range which do not fulfil the precondition of its neighbourhood being greater or equal than κ are marked as "density-reachable"

points. Those points are not part of the original cluster but in range so the point is considered to be better than pure noise because of a cluster being in its ϵ radius.
This process loops through all data sets, finds the clusters and continues until every point is marked as "visited" and classified as either "core", "density-reachable" or "noise".

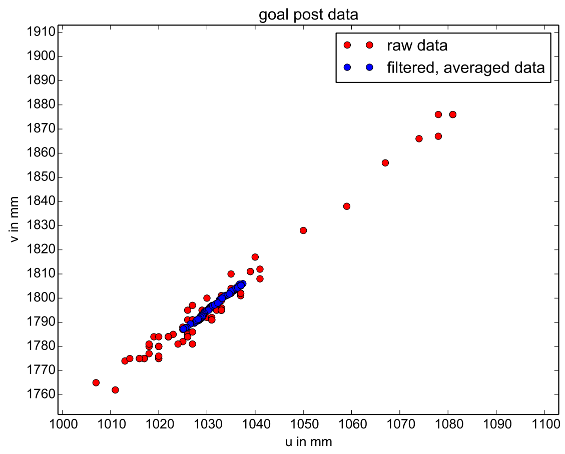

Fig. 3: Plot of DBSCAN-filtered and averaged data. 100 data sets have been recorded in about 5 seconds. Original vertical distance (v) to the post was 2000mm and horizontal distance 1000mm. The robot was standing still.

Throughout our testing process we experienced positive results with a κ of 6 and an ϵ of 50 as parameters for our DBSCAN implementation.
After our DBSCAN has processed all data it returns the found clusters and density-reachable points. Those data is not directly returned to modules requesting filtered data. Before sending the filtered data our filter module calculates the arithmetical mean for every cluster plus density-reachable points.
Figure 3 shows a dataset of 100 red unfiltered points and the filtered and averaged blue points. The euclidean distance between the filtered points is decreased heavily and outliers are reliably marked as noise and therefore not processed nor averaged.
Nevertheless the goal post information may be sufficient for some purposes, but at least for the goal center calculation this is not sufficient. Due to calibration issues and the distribution of noise the goal center calculation would not be optimal then calculated out of this data directly.
For this purpose, to further improve the stability and precision of goal center information, we implemented an artificial neural network.

Artificial Neural Network For fixing the calibration issues and more accurate data our idea was to implement an artificial neural network (multi-layer perceptron design).
The module is designed object oriented and caches every calculation to prevent from something being calculated twice. This, again, increases the space complexity but decreases the computing time and was simpler in cases of debugging, too.
In addition to this the whole data (weights, bias, activations, outputs, ...) is stored in arrays of Numpy and always calculated with Numpy matrix operations or scalars to save as much computation time as possible.
Furthermore the neural network is designed dynamically. All configurations are variably. At the moment our net has an input layer with 2 neurons, 2 hiddens layers with 10 and 5 neurons and an output layer with 2 output neurons. The learning rate ϵ is 0.2 with 10.000 learning steps.
All this configurations can be changed instantly, for example a list of integers $[10, 5]$ represents the hidden layers at the moment and can be changed to $[20, 10, 5]$ which would create a net with 3 hidden layers with 20, 10 and 5 neurons per layer.

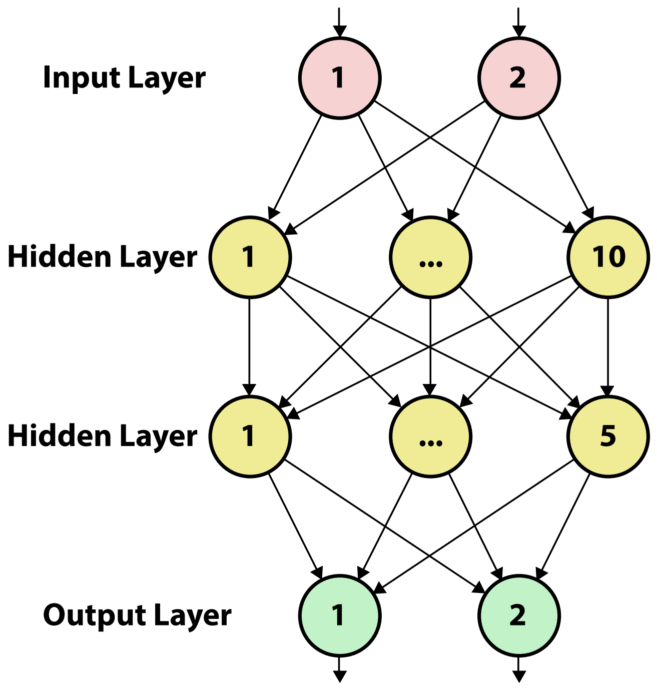

Fig. 4: Plot of our Artificial Neural Network design (multi-layer perceptron).

2.4 Localisation

Calculating our robot's position The robot provides a lot of different data. But unfortunately a most of the values are not fixed points on the field and so they are unsuitable to calculate the robot's position. From the data given by the robot only the distances to the goal posts are sufficient for our purposes.
But the information the robot actually provides is not the direct distance but only the distance in front of the robot and the distance to the left of the robot to the objects. We will call the distance in front of the robot u and to the left of it v to stay consistent with the terms used in the code. figure 5 contains an example of these values.

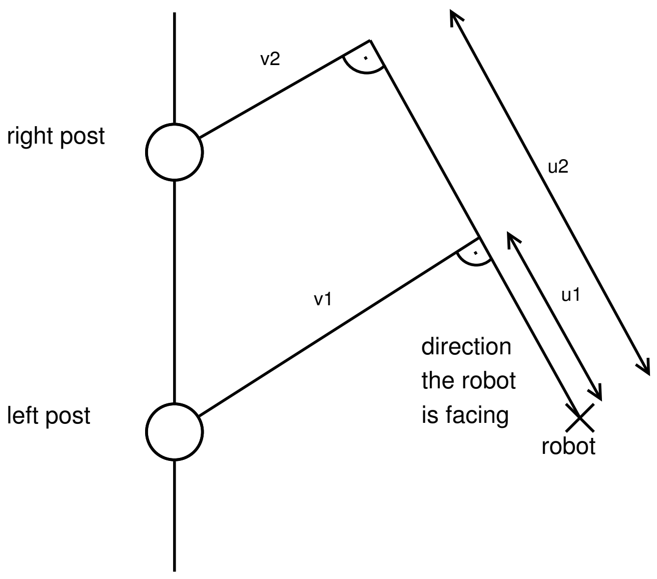

Fig. 5: Example of u- and v-values.

At first we want to outline how we basically calculated the position of our own robots and afterwards we are going to explain all steps in detail.
For this calculation we used the u- and v-values to two goal posts. We use these values to calculate the direct distances to the goal and get two lines that intersect exactly at the robot's location. For this reason it's necessary to calculate the slope of the two lines so that we can form equations for them. With these we can determine the point of intersection. And once we know the point of intersection we also know the robot's position.
We are going to call the u- and v-values for the left post u_1 and v_1, and for the right post u_2 and v_2. You can use Pythagora's theorem to calculate the direct distances to the goal posts as done in equation 1.

$$d = \sqrt{u^2 + v^2} \tag{1}$$

We do that for both goal posts and we are going to call the distance to the left post d_1 and to the right post d_2. We want to know the angle between these lines and the goal width ψ so we need to know this value as well. The goal width doesn't change and we can retrieve it in the code, but the robot's goal post recognition can be very inaccurate. Meaning even though in reality the goal width is always the same, according to the robot's values the goal width is changing all the time. So the robot calculates it's own value for the goal width too. We use the u- and v-values to create vectors and calculate a vector between the two goal posts:

$$\mathbf{gw} = \begin{pmatrix} u_1 \\ v_1 \end{pmatrix} - \begin{pmatrix} u_2 \\ v_2 \end{pmatrix} \tag{2}$$

The absolute value of gw is the goal width ψ.
Now that we computed these three values we can form a triangle like in figure 6.

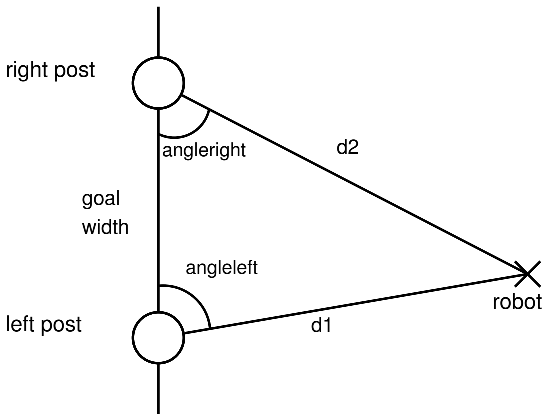

Fig. 6

With the help of that triangle we are going to calculate the point of intersection of d_1 and d_2. Let's assume we have a coordinate system and the point of origin is at the middle of the field. The reason for that is that the origin of our grid world, that is used to save the data, is in the middle of the field too. That means it's easier to translate the position found in that coordinate system to the grid world. But we will explain that in the next section. Otherwise it would have been easier to have the origin in the middle of the goal the robot is facing to.
To calculate the point of intersection we need to form two equations for d_1 and d_2. In order to do that it's necessary to know the slope of both lines.
We calculate the two angles shown in figure 6. The formula we used is the following:

$$\alpha = \arccos\left(\frac{a^2 - b^2 - c^2}{-2bc}\right) \tag{3}$$

Specifically these are the two formulas if you want to calculate the angle of the left post and the right post in the triangle:

$$angleleft = \arccos \frac{(d_1^2 - d_2^2 - \psi^2)}{(-2 * d_2 * \psi)} \tag{4}$$

$$angleright = \arccos \frac{(d_2^2 - d_1^2 - \psi^2)}{(-2 * d_1 * \psi)} \tag{5}$$

To convert the angles to the slopes of the two lines in the coordinate system, we can do the following:

$$sloperight = \frac{1}{\tan angleright} \tag{6}$$

On the left side we don't need the angle itself but 180 minus the angle:

$$slopeleft = \frac{1}{\tan (180° - angleleft)} \tag{7}$$

Because we want to know the point of intersection of these two lines we need an equation for them:

$$y_1 = slopeleft * x + b_1 \tag{8}$$

$$y_2 = sloperight * x + b_2 \tag{9}$$

b_1 and b_2 are the points of intersection with the y-axis. Our coordinate system has it's point of origin at the middle point of the field. The reason for that is that the origin of our grid world is at that point as well. So converting the values is a lot easier. The unit of the point of intersection is now millimetre, but we convert these values to enter the robot's position into our grid world.
We know the x- and y-values at the points of the goal posts. It is half the field length for both x-values and half the goal width for y_1 and minus half the goal width for y_2. With these points we can determine the values for b_1 and b_2 by rearranging the formulas. Afterwards we equate the equations 8 and 9 and solve this new equation for x. Then we can enter the found x-value in one of the two formulas to determine y. The found x- and y-value is the position of the robot in the coordinate system.

Now the robot knows it's own position. But it can't determine the opponent's position on the field yet, because it doesn't know which direction it faces. The robot has its own inner coordinate-system. 0,0 is the middle of the robot. The orientation of the y-, and x-axis depend on the direction the robot faces.
So at first we calculate how many degrees the robot would have to turn to face one of the two goal posts. If the robot turns to that angle and would walk directly to the goal post he would end up directly in front of it, but not looking directly at it. To achieve that we need another angle. Now if the robot would turn that angle he would look directly at the goal post. So all we need to do at

the end is, according to the exact situation at hand, either add or subtract the two angles we calculated and we would know the angle to the x-axis. In figure 7 the first angle we mentioned is β_2 and the second one is newangleright. β_1 and

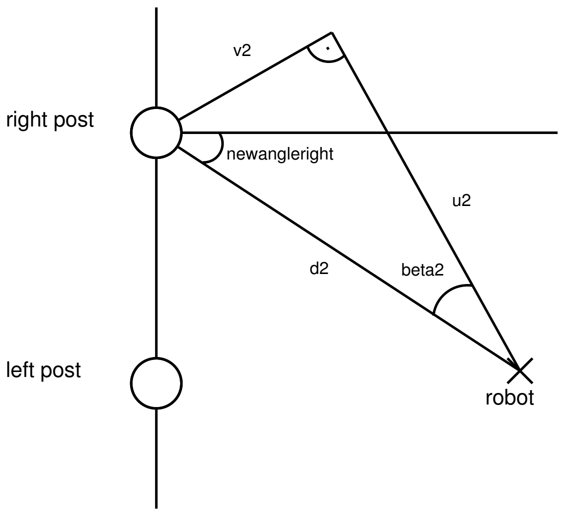

Fig. 7: Example of angles needed to calculate the angle to the x-axis of our grid

β_2 are calculated with the following equation:

$$\beta = \arcsin\left(\frac{v}{d}\right) \tag{10}$$

We explained earlier how we calculated angle_right and angle_left. angle_left_new and angle_right_new are just 90°minus the old angles. This only works if the old angle is under 90 degree, but this is always true for at least one of the two angles.
Now there are different cases which affect how exactly we calculate the angle to the grid. At first we check if u1 or u2 is bigger. In our example we face the opponents goal. If u1 is bigger that means that the robot faces to the left and that the angle is bigger than 180°. In that case we have to add 180°and in the other case we have subtract from 180°. It also determines if we take angle_right_new or angle_left_new depending on the side the robot is facing. Meaning if u1 is bigger we use angle_left_new and if u2 is bigger we use angle_right_new.
The v-values are also important. With these values we can determine if the triangle of the u- and v-values is inside or outside the triangle of the two goal posts. So depending on that we need to either add beta and angle_new or subtract beta from angle_new.
So now that we know the position of the robots and their orientation we can finally calculate the position of other robots on the field.

Calculating other robot's positions When our robot sees another robot it gets it's distance as u- and v-values. Once again we can calculate the direct distance by using equation 1. We want to use polar coordinates to calculate the position of the other robots. These consist of the direct distance r to the object and the polar angle ϕ, which is the angle between r and the x-axis of the grid in our case. In order to calculate ϕ we need to know the angle from u to r. We use equation 3 for this purpose, and substitute u for a, v for b and r for c.
We already determined the angle from the direction the robot is facing to the grid.
To finally get the angle from the object to the grid we have to either add or subtract these two angles depending if the robot is left or right of the object, meaning if the v-value is positive or negative. So depending on the circumstances we use either equation 11 or equation 12.

$$\phi = (angle\_robot\_to\_grid - angle\_robot\_to\_object)\%360 \tag{11}$$

$$\phi = (angle\_robot\_to\_grid + angle\_robot\_to\_object)\%360 \tag{12}$$

Afterwards we use equation 13 and 14 to determine the x- and y-values.

$$x = r * \cos(\phi) \tag{13}$$

$$y = r * \sin(\phi) \tag{14}$$

These values still depend on the position on the robot, so at the end we have to add the position of the robot to the found values to finally get the position of the other robot on the field.

Grid World Now that we know the positions of our own and other robots we need some way to save and share these positions, so that our team members can make use of that knowledge and every robot in our team can profit. We needed a representation of the field, so we could store and use the data we generated. For that purpose we use the grid world. Other approaches we considered were a coordinate system that uses millimetres and a tree based solution [3]. In the end we decided to use said grid world because of its simplicity and several advantages. Even though Daniel had the idea for that representation of the field it turns out we are not the first robocup-team to use it. [2] So we need to divide the field into a certain number of cells. As shown in figure 8 the origin of the cells is at the midpoint of the field. We had different iterations of the grid world with different origins. At first the origin cell was in the lower left corner. Then we decided it would be better if the origin cell were in the middle of our own goal, mostly because it made calculations needed to determine the position of the robots easier. But if we wanted to use these calculations for both goals, it is easier to have the origin cell in the middle. There is also a difference between choosing an even number and uneven number respectively on the x and the y direction. We decided to choose an uneven number for both, which means the origin cell is now in the middle of the centre spot. One benefit of that is that there is a cell that is exactly in the middle of the goal where the goalie stands

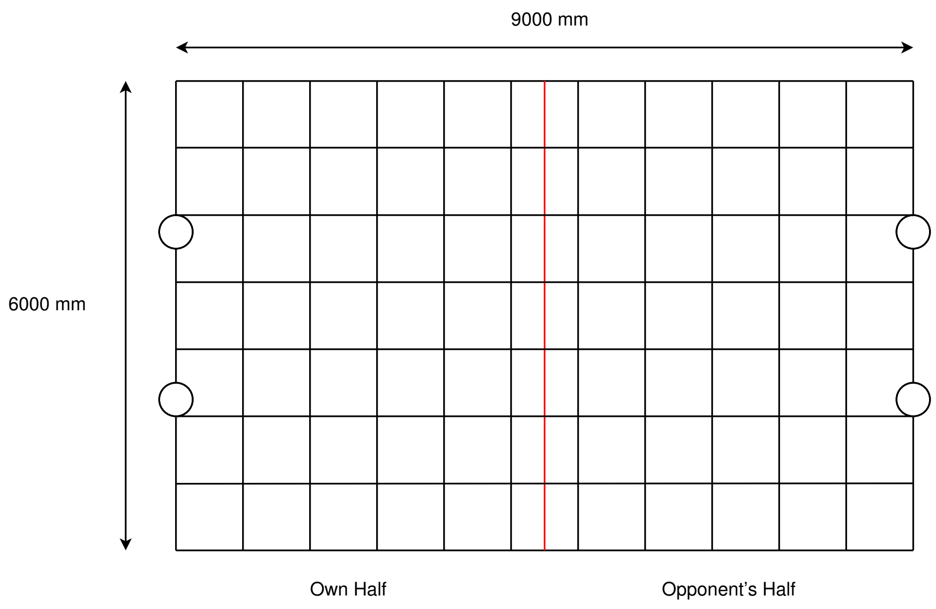

Fig. 8: The grid world divides the field into a certain amount of cells.

most of the time.
Also when we take the cell at position (0,0) for example and we compare it to a similar coordinate system, then the upper right edge of that cell would be at (0.5, 0.5) in the coordinate system. When we calculate positions we don't get integers but decimal digits. So when we determine a robot's position on the field, we can simply round that result and know into what cell we have to put it.
We explained earlier how to calculate the robot's position, but all the results were in millimetre. To convert these values into a cell one can use equations 15 and 16.

$$x\_cell = \frac{x\_mm}{field\_length} * x\_cell\_count \tag{15}$$

$$y\_cell = \frac{y\_mm}{field\_width} * y\_cell\_count \tag{16}$$

To store the data we use a dictionary in Python. The indexes of the dictionary are the positions in the grid world. For example (0,0) would be the centre of the field. Then we use numbers to fill that dictionary. At initialization the dictionary will be filled with zeroes, which is representing that there is nothing at that particular position. Here is a quick overview of the numbers we fill the dictionary with and what they mean:
0: nothing
1: own goal
2: enemy goal
3: own position

4: team mate
5: opponent
6: unclassified obstacle
7: ball

So if we want to know the state of a certain position we call that dictionary and get back a number between 0 and 7.
Every robot can determine their own position themselves. The position of the team members is shared over the wi-fi, which will be explained in the next section. The robots can determine the position of opponents that they see. They share those positions so everyone has the position of as many opponents as possible. The robots actually can't distinguish between team mate and opponent. But because we know all the positions of our team members the robots can compare the position of the robots they see with the location of their team mates in their grid world. Only if there isn't a team mate in that cell they will identify the robot as an opponent.
This is one of the advantages of this representation of the field. The inaccuracy makes it easier to compare the found objects on the map between team mates. If we would measure everything in millimetre, every robot would probably calculate slightly different positions for the objects and so comparing them would become more difficult.

2.5 Communication

Communication has always been one of the most important fields of resaerch in RoboCup, at least it was the may reason to found the instituition to find out whether two or more robots might posibilly be able to act together in a sensful way. Due to the increasing number of robots one team is allowed to have on the field (up two eleven in 2050, as the roadmap plans) and the high costs for one robot it will become more and more necessary to cooperate with other teams. When there were the first tries of mixed teams some years ago, it was very fast obvious, that robots from different teams will have to communicate with each other. So some years ago, a RoboCup Soccer Humanoid Kid Size team implemented a WiFi communication standart, the so calles MiteCom, that is used by a lot of taems in this league and makes it very easy to to play in one team with robots from different teams. This standart already was used in our team for very simple behaviour. for example because of the lack of localisation, it was not possile for a fiel player to determine whether a goal was the opposite goal or his own goal. So the goalie could send information like "The ball is very close to me" and thus make the field player stop playing on that goal, but turning around and trying the ohter goal. But this was a very unsecure mechanism, because the WiFi on competitions has a very high package loss and is very unreliable. So our idea was, that every robot could provide its own gridworld and in return receive every other robots gridworlds. By weighting the incoming data due to reliability (like taking the median for each position of ten or so incoming grids) or actuality

(I did not see a goal for a long time, dont know whether my own position is right or not, i will take a new grid and search a goal) the average information quantity and quality could increase a lot. But our a big problem was, that via MiteCom it is only possible to send 32 bit integer values. Foremost we thought about implementing our own standart, but when thinking about it a secound time, it seemed a good idea to use it. First of all, because of the aspect of universality mentioned above, that will be needed in the future. But forthermore it is not a bad idea to only send integers. As already mentioned, on competitions the quality of WIFI is not very hight. So the big package loss could cause incompletely received data, and so only sending "small" 32bit integers is more practicable. So the problem appeard, how to encode our dictonary, that represents the grid world into an integer. Our solution is based on binar numbering system. Each of the 32 positons represents one cell in the grid world. But dividing the hole $54m^2$ field in only 32 cells seemed very few. So we use 8 integers, each representing one eighth of the field. Besides, the binary integer can only depict one kind of information (numbers 1-6, see Gridworld) at a time. So we also need one integer per information type. When calculating these integers, the current part of the grid is scanned for the certain kind of information. When a gridcell is found, the gris cells number becomes the exponent of 2 andthe sum of all of them gives the 32bit integer, that is thent via MiteCom.

3 Results

Present your results in a logical sequence

Highlight the importance of your results and explain your analysis methodology

Discuss the results you infer from your work

Important: Adopt a critical approach in your discussion

Do not oversell your results - put the advantages first, but dont forget to mention the shortcomings!

4 Summary

Be more informative than your abstract!

Include a concise version of your discussion

Highlight what you found out

Highlight the problems you encountered

Explain how your results support your conclusions!

Provide suggestions for future research and briefly outline how suggested research can be attempted

Important: Make this section readable

5 References

Very important section of your report

If you used external information/results) Provide a reference!

References will help the reader understand the basis of your work and provide context for comparison

Use of references might also help you to be more concise

There are several types of reference

Book

Journal article

Conference publication

Web site

Web sites are usually unchecked sources - be careful

Notes and Comments. The first results on subharmonics were obtained by Foster and Kesselman in [6], who showed the existence of infinitely many subharmonics both in the subquadratic and superquadratic case, with suitable growth conditions on H'. Again the duality approach enabled Foster and Waterman in [8] to treat the same problem in the convex-subquadratic case, with growth conditions on H only.

Recently, Smith and Waterman (see [4] and May et al. [5]) have obtained lower bound on the number of subharmonics of period kT, based on symmetry considerations and on pinching estimates, as in Sect. 5.2 of this article.

References

1. Ester, M., Kriegel, H.-P., Sander, J., Xu, X.: A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In: Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96), pp. 226–231. AAAI Press, Munich (1996)
2. Salehi, M.E., Safdari, R., Abedi, E., Foroughi, B., Salimi, A., Farokhi, E., Teimouri, M., Shakiba, R.: MRL Team Description Paper for Humanoid KidSize League of RoboCup 2014. Mechatronics Research Lab, Dept. of Computer and Electrical Engineering, Qazvin Islamic Azad University, Qazvin, Iran (2014)
3. van der Molen, H.: Self-localization in the RoboCup Soccer Standerd Platform League with the use of a Dynamic Tree. A Bachelor Thesis in Artificial Intelligence, University Of Amsterdam (2011)
4. Smith, T.F., Waterman, M.S.: Identification of Common Molecular Subsequences. J. Mol. Biol. 147, 195–197 (1981)
5. May, P., Ehrlich, H.C., Steinke, T.: ZIB Structure Prediction Pipeline: Composing a Complex Biological Workflow through Web Services. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 1148–1158. Springer, Heidelberg (2006)
6. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco (1999)

7. Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C.: Grid Information Services for Distributed Resource Sharing. In: 10th IEEE International Symposium on High Performance Distributed Computing, pp. 181–184. IEEE Press, New York (2001)
8. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid: an Open Grid Services Architecture for Distributed Systems Integration. Technical report, Global Grid Forum (2002)
9. National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov