RoboCup Thesis Report

ROBOT DETECTION USING BAYESIAN MACHINE LEARNING

August 25, 2014

By Jaiden Ashmore, z3330870

The School of Computer Science and Engineering

Supervisors: Dr. Bernhard Hengst, Dr. Maurice Pagnucco

Assessor: Dr. Claude Sammut

Acknowledgments

I would like to thank Bernhard Hengst for all the guidance that he gave me in completing my thesis. Throughout the entire thesis he provided great advice and guidance which directly influenced the eventual completion. There were numerous occasions when I was stuck or going down a wrong path and every time Bernhard was able to see the forest from the trees and guide me back in the right direction.

I also want to thank the 2014 development team for all the hard work they put in to make us the SPL world champions. Everyone was living off little sleep, stress and pizza but were still able to smile and crack jokes which made the experience far more enjoyable.

Next I would like to thank my family for all the support that they have provided over the weeks leading up to the competition. They were always there to help me out, provide me with food or lifts, which I greatly appreciated.

Abstract

The ability for a robot to see other robots on the field is a crucial component in ensuring that fair play is upheld throughout the match. Running into robots causes the player to be penalised and removed from the field so a high quality robot detection and avoidance algorithm is needed. This year the current robot detection and avoidance code was rewritten with these goals in mind.

The detection code was rewritten to be more modularised, therefore allowing easier extensions for the future. A Bayesian machine learning algorithm was applied to the detected robots to improve true positives and true negatives. The robot filter was also rewritten to perform better with the detection set as the current implementation was not performing well in practise. Finally the robot avoidance was modified to include avoidance of far-away robots, as well as tuning close robot avoidance.

Contents

Chapter 1

Introduction

1.1 Overview

The RoboCup is an annual competition that is used to present research advances in fields such as vision and artificial intelligence, through challenges such as robots playing soccer and robot rescue. While the goal of the teams is to win the competition, the fruits of their labour are important in expanding research that can be applied to real world applications.

1.1.1 Justification

The ability for the robot to be able to see and strategise against opponent robots is crucial in a winning team's algorithm. The more information available on the opponent's location, the better decisions that can be made and therefore give the team a better chance of winning.

With the penalty increase to 45 seconds for robot pushing the ability to detect and avoid close robots is important in keeping players on the field. If the robots were to be constantly bumping into the robots, many would be off the field at once, which is a huge disadvantage.

Being able to detect far away objects also helps the robot in being able to make preemptive decisions to avoid the robots. They can have the ability to change their path to still get to the destination but go via the optimal path that avoids all robots.

Gameplay decisions can also be applied with greater success if the opponent's location is known. For example, if the robot has possession of the ball but detects a robot nearby, it can quickly make a decision to dribble away instead of kicking as this is faster and will allow it to maintain possession.

1.1.2 Goal

With the need for improved robot detection and avoidance there are specific goals needing to be achieved:

- **Improved far-away detection**: the new robot detection should work efficiently for robots up to three metres away, if not more. This would be an improvement on the current code which is better tuned for close robots.
- **Improved close detection**: a higher success rate for detecting close robots is essential in order to avoiding pushing other robots. This is particularily important due to the increased penalty for pushing.
- **Improved jersey detection**: it should be able to detect jerseys of the robots even when very little is shown. This detection should be modularised particularily for when RoboCup changes the jersey rules and allows team's own customised jerseys.
- **Less reliance on colours**: the code should work better when there are bad colour calibrations or lighting. Even if there are bad colours on the robot it should still be able to detect the robot.
- **Better grouping of robots when moving:** when the robot is detecting multiple robots over a time-frame the algorithm must be able to determine which are matching robots between frames. E.g. if there are two robots in one frame and two in another that are in a slightly different position, the algorithm must be able to distinguish this and create two distinct obstacles for the robot to be able to avoid.
- **Able to see far-away robots and avoid them:** the robot should be able to identify robots between it and it's objective and alter it's vector to avoid it.

1.1.3 Methodology

In completing this thesis an agile methodology was used to allow for quick iterations that result in workable products throughout the entire development process. This allows for quick testing and always having a working model that can be used on the robots.

The ability to test often allows for quick experiment and modification to the code. If a big bang approach was used, it is unlikely to work as intended and it is possible to run out of time as the problems may take too long to fix. Iterative testing through the agile methodology fixes this as there is always a working model which can guide the development to the final solution.

1.1.4 Outline

Chapter 3 - Robot Detection: talks about how the robot detection code was modified to improve overall detection, especially far away detection. It also delves into the modularisation of the algorithm to improve extensibility and maintainability.
Chapter 4 - Robot Filter: explains the need for a new robot filter and the implementation that was used to achieve this.
Chapter 5 - Robot Avoidance: describes how the robot detection and filter code is used to avoid robots that are on the field, therefore reducing penalties due to pushing.

1.1.5 Definitions

Word	Definition
Frame	This is a visual frame that is captured. Over a course of a second the robot's camera may capture 20 of these which are processed
Saliency Image	After capturing the raw image, this image is converted into distinct colours representing field, background, ball, goal, white, etc. This simplified colour image is the saliency image.
Possible Robot	This represents a currently detected robot that could possibly be an actual robot. This term is used in the new robot detection and once it finishes the detection process the possible robots become visual robots
Field of view	This represents the angle of vision that the camera can see. E.g. the robot's camera may be able to see 60°horizontally, 30°left and 30°right.
Heading	This represents the angle from the robot to another point on space. E.g. a heading of 30°represents 30°to the left, whilst -30°represents 30°on the right.

Chapter 2

Background

2.1 Introduction

This chapter includes information about background research that was completed before and during the production of the thesis project. It mostly involves researching how other implementations for the robot detection work to figure out ways to improve upon them.

2.2 Literature Reviews

2.2.1 2011 rUNSWift Robot Detection report

In 2011 rUNSWift implemented a robot detection code[1] algorithm which looks for obstructions in the field edge and then uses machine learnt data to construct a decision tree for sanity checking detected robots.

What it does is it looks along the field edge and finds the first green point below it, as seen on the left of Figure 2.1. From there, it binds the dips into groups that are likely to be robots, removing any small unimportant dips. It then goes through processes of estimating the location of the head of the robot, finding the waistband to determine the team and sanity check detected robots using the decision tree and other checks. The result of this process can be seen on the right of Figure 2.1.

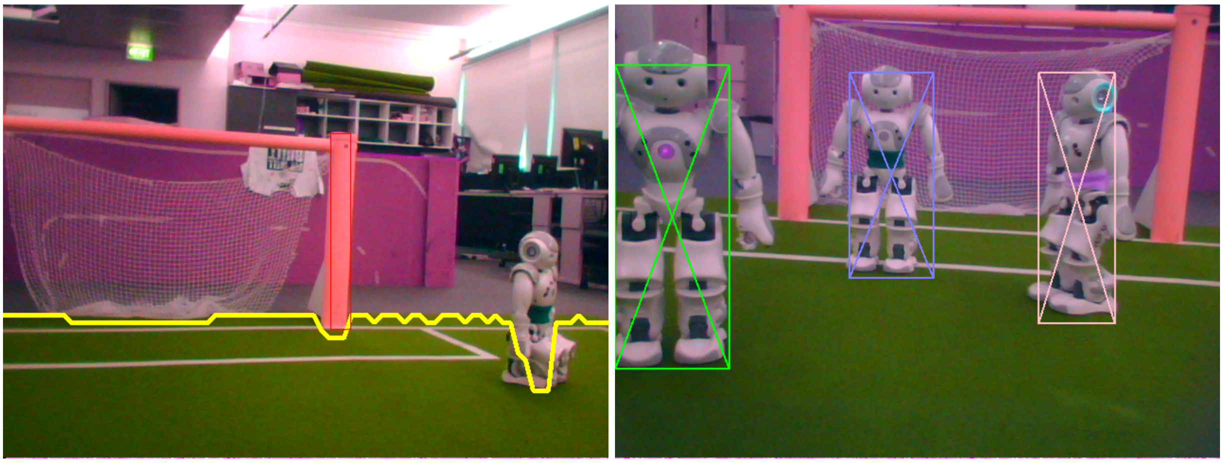

Figure 2.1: 2011 Robot Detection - obstruction detection

Note: this code was not included into the 2011 RoboCup competition due to lack of processing power to run the algorithm.

2.2.2 2013 BHuman Obstacle Detection

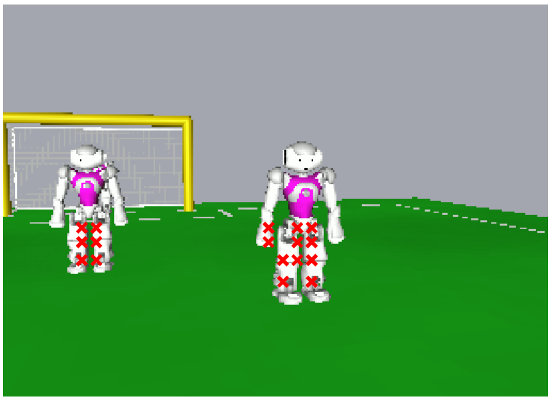

Figure 2.2: 2013 BHuman obstacle region detection

The BHuman's obstacle detection code was described in the 2013 code release[2] and is very different to the detection methods described above. It looks on the field and marks any regions that are significantly non-green and are larger than a field line. Figure 2.2 shows the regions that were detected in this frame. It takes these points and determines where the possible bottom of the obstacle is. This will likely detect both feet and hands so it tries to remove any hands from the image as they do not consider them to be useful. Afterwards the height of the robot is estimated as being the first region of significant green or until it reaches the horizon. Here the top and bottom camera's detections are stitched together and groups of regions are merged together to create the robot obstacle.

2.3 Common Ideas

In both of the obstacle detection algorithms, the code has the job of collecting possible obstacles and, from this, trying to estimate where the feet and head of the robot would be. The main difference between these two algorithms is where it begins to look. The rUNSWift code uses the field edge to determine where to begin, whereas the BHuman code looks on the whole field to determine possible robot groups. Both do similar tasks in determining where the bottom of the robot is, and both estimate the top of the robot using either the horizon or trigonometry.

pre

2.4 Relevance

Both of these examples shows ways that robots could be detected and so are a good foundation for finding ways to improve on them in a new implementation. As they both represent methods that are quite dissimilar in their process an algorithm that is the better of both worlds could be implemented.

2.5 Emergence of necessity

The 2011 rUNSWift robot detection code was not able to be included into the competition as it was not able to run on the processor. With improvements in the processor and camera in robots bought since 2011, an algorithm that is able to utilise this processor, while still not overpowering it, is needed. The algorithm written in 2011 has also been heavily modified over the years with little to no documentation, so a new algorithm that is well documented, modularised and performant is needed so future maintenance can be done easily.

We wanted to base our robot detection on the foundations of our previous implementations instead of copying BHuman's detection. Having the desire to write robot detection that was better than theirs was a good reason to analyse BHuman's process.

2.6 Conclusions

In conclusion, we wanted an algorithm that had as it's basis the rUNSWift's previous robot detection code which we could improve for better success rate and higher maintainability. An algorithm that was also able to beat BHuman's robot detection code was particularilly desirable.

Chapter 3

Robot Detection

3.1 Introduction

The robot detection algorithm is designed to use vision information, as well as sonar, to be able to detect robots on the field, as well as determining what team they are on.

While explaining the algorithm in the following sections the frame seen in Figure 3.1 will be used.

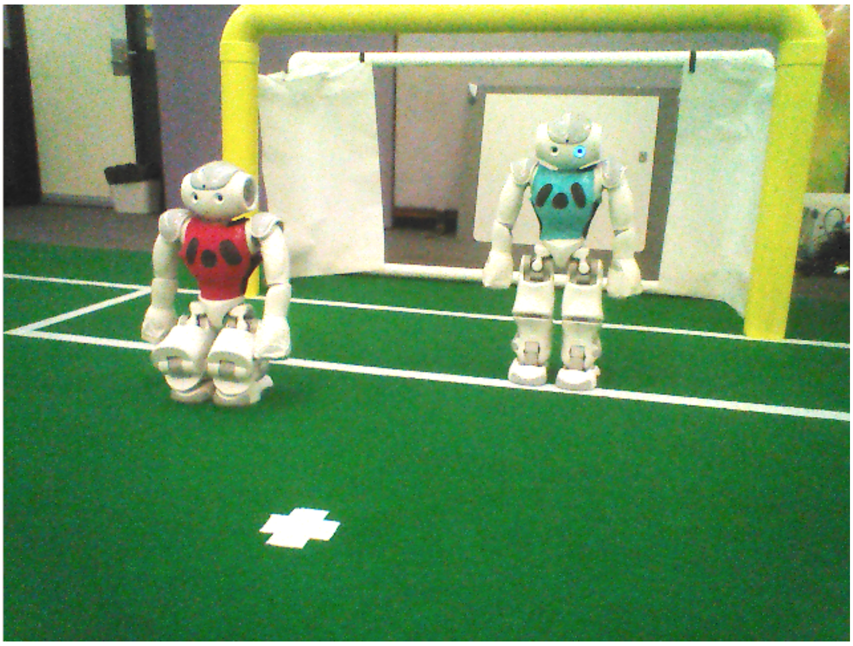

Figure 3.1: Frame used to demonstrate robot detection

3.2 Implementation

3.2.1 Determining Field Edge Obstructions

Goal

The goal of this step is to look at the field edge and determine if there are any points along the line where the green field can be seen. The assumption has been made that if the field cannot be seen at this point there must be something blocking it, e.g. an obstruction/obstacle. For example, in Figure 3.2 there are multiple points where the field cannot be seen and they are marked as obstructions via the blue lines.

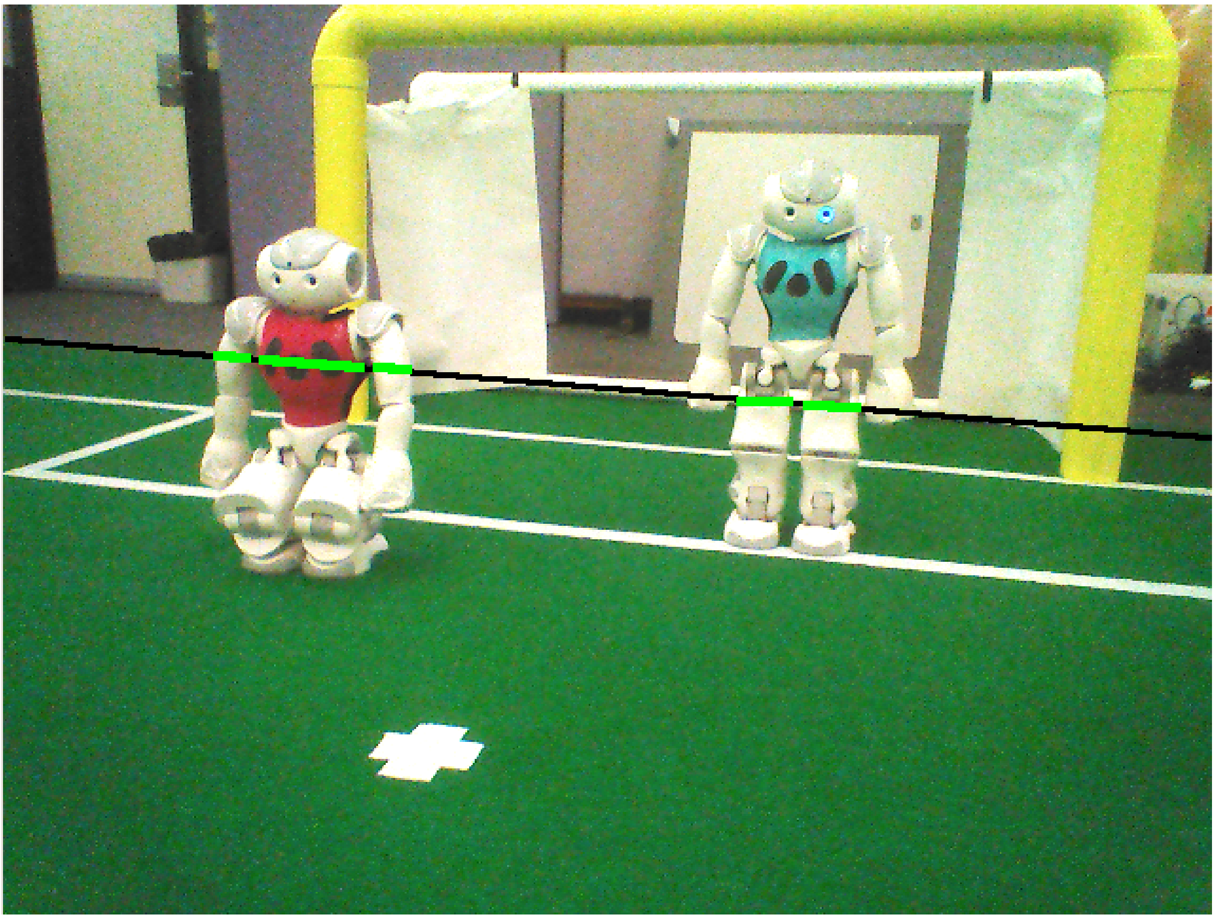

Figure 3.2: Goal for obstruction detection

Algorithm

The field edge algorithm has determined a set of points representing the field edge and this will be used to determine obstructions. At each column of the field edge it uses $\underline{x}$ pixels below the field edge to determine if it is an obstruction. For example, in the current implementation, it uses 5 pixels below the field edge. For each pixel a score is given depending on what type of pixel it is. e.g. field: -3, white: 3, background: 1, etc. If the accumulated score is above 0 it is assigned as an obstruction.

The reason for using a scoring system was to give more weighting to certain pixels. If there were only background and field pixels in the column, it would not necessarily want to be considered as an obstruction as strongly as if only white pixels were seen.

Another approach could have been to only consider the pixel row as field if all the pixels were green. For this to work correctly the lighting conditions, camera settings and colour calibrations would have to be perfect, a configuration quite unachievable. Therefore, it is common for the field to have white pixels or a robot to have green pixels so the scoring system allows for margins of error in the camera.

After each column has been assigned as either field or obstruction, they are grouped into sections. This is a simple loop that goes over each of the points and if there is a certain threshold of consecutive obstruction columns (e.g. 5 pixels) they are merged. Also, if there are small gaps between obstructions they should be merged to create larger obstructions. These gaps can be caused by bad pixel colours, such as green on a robot.

The output of this algorithm is a list of lines (2 points), representing possible obstructions.

Result

As can be seen in Figure 3.3, the algorithm did well in detecting the intended obstructions in the example frame used.

The algorithm was further tested on a larger sample set and the data seen in Table 3.2.1 was collected. For the purpose of data collection, the following criteria was used to cal-

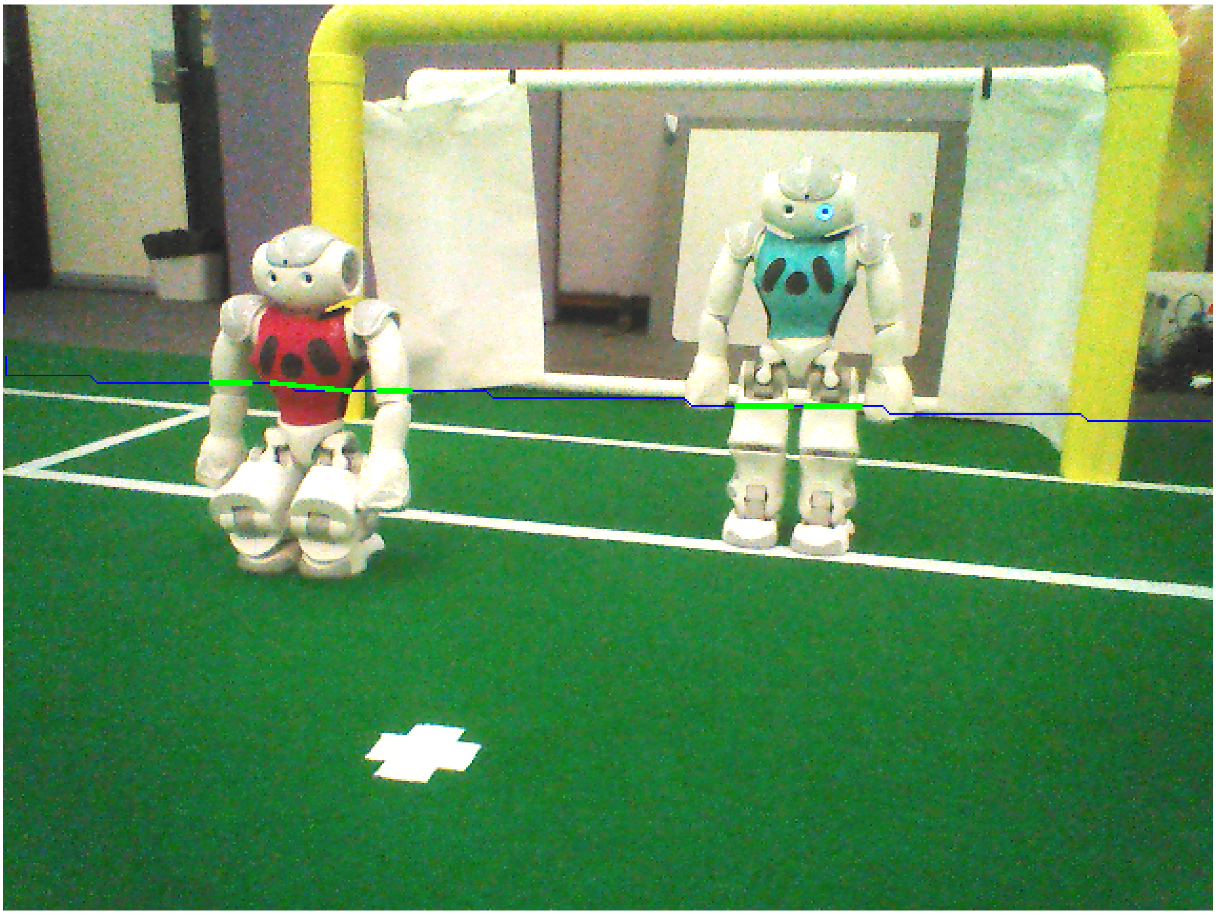

Figure 3.3: Result for obstruction detection

	Is an obstruction	Is not an obstruction	
Detected as obstruction	54	10	54
Not detected as obstruction	17	56	73
	71	66	137

culate true positives, etc. These can be seen visually in Figure 3.4.

$$\begin{aligned} Sensitivity &= \frac{\text{True positives}}{\text{True positives} + \text{False Negatives}} \\ &= \frac{54}{54+17} \\ &= 76.0\% \end{aligned} \tag{3.1}$$

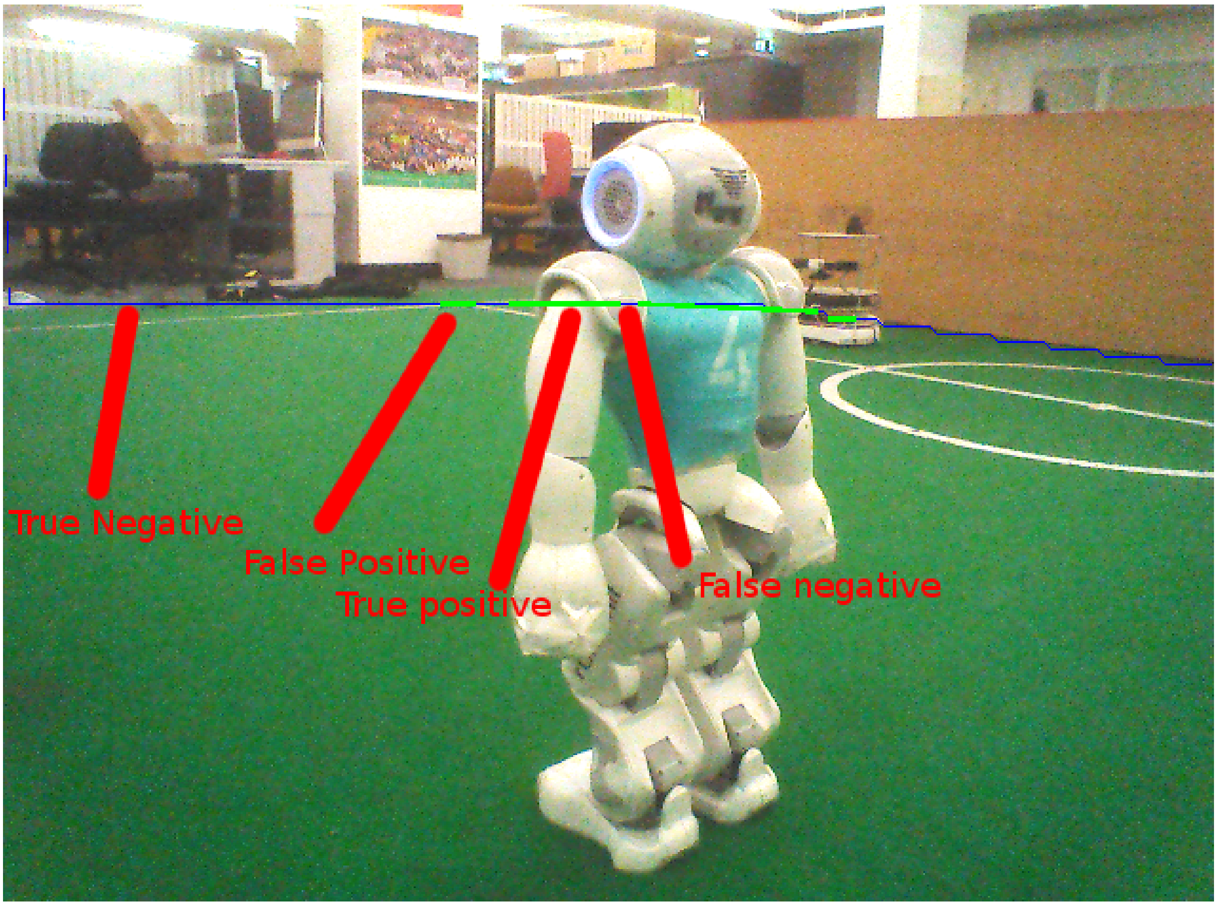

Figure 3.4: Example of true positives, false positives, etc in obstruction detection.

$$\begin{aligned} Specificity &= \frac{\text{True negative}}{\text{True negative} + \text{False positive}} \\ &= \frac{56}{56+10} \\ &= 84.8\% \end{aligned} \tag{3.2}$$

Analysis

From the results, it can be seen that the success rate for non-obstructions does well with 84.8% specificity. The success rate for detecting the obstacles does work however with only a 76% success rate. The reason for this lower success rate is mostly due to gaps in-between observations due to bad calibration of the blue jersey. This failure can be seen in Figure 3.5 where the blue jersey has been detected as green and therefore the algorithm thinks that it is field when it is not. This should hopefully be fixed later in the robot detection where detections are merged, so should is not be a problem.

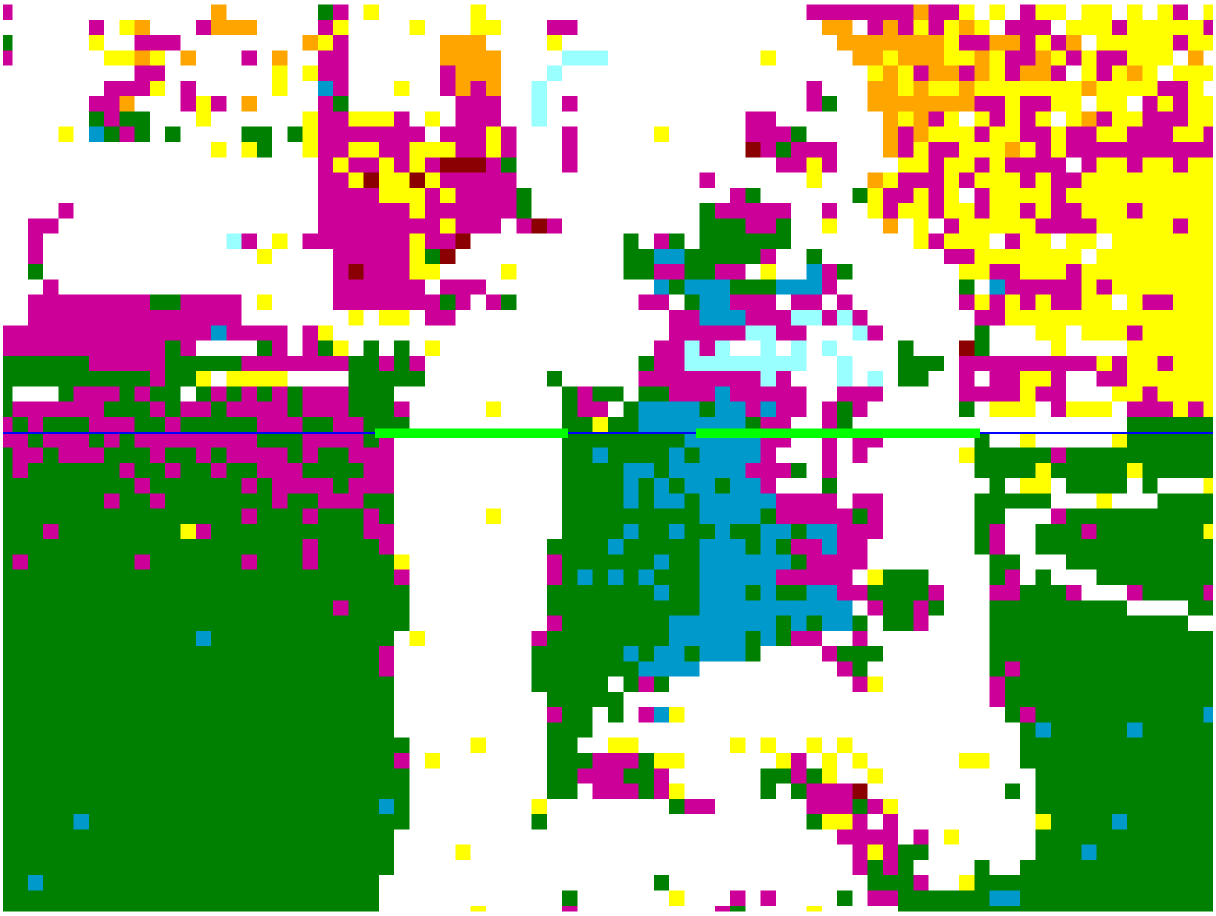

Figure 3.5: Example of bad calibration affecting observation detection

3.2.2 Searching for the bottom of obstructions

Goal

The goal of this step is to take the possible obstructions and to extend it down to the bottom of the obstruction, e.g. the feet of the robot. This can be seen in Figure 3.6 where a bounding box representing the legs and bottom of the arms for the obstruction has been created. This should work even if the colour calibration is not optimal and should quickly complete the calculation.

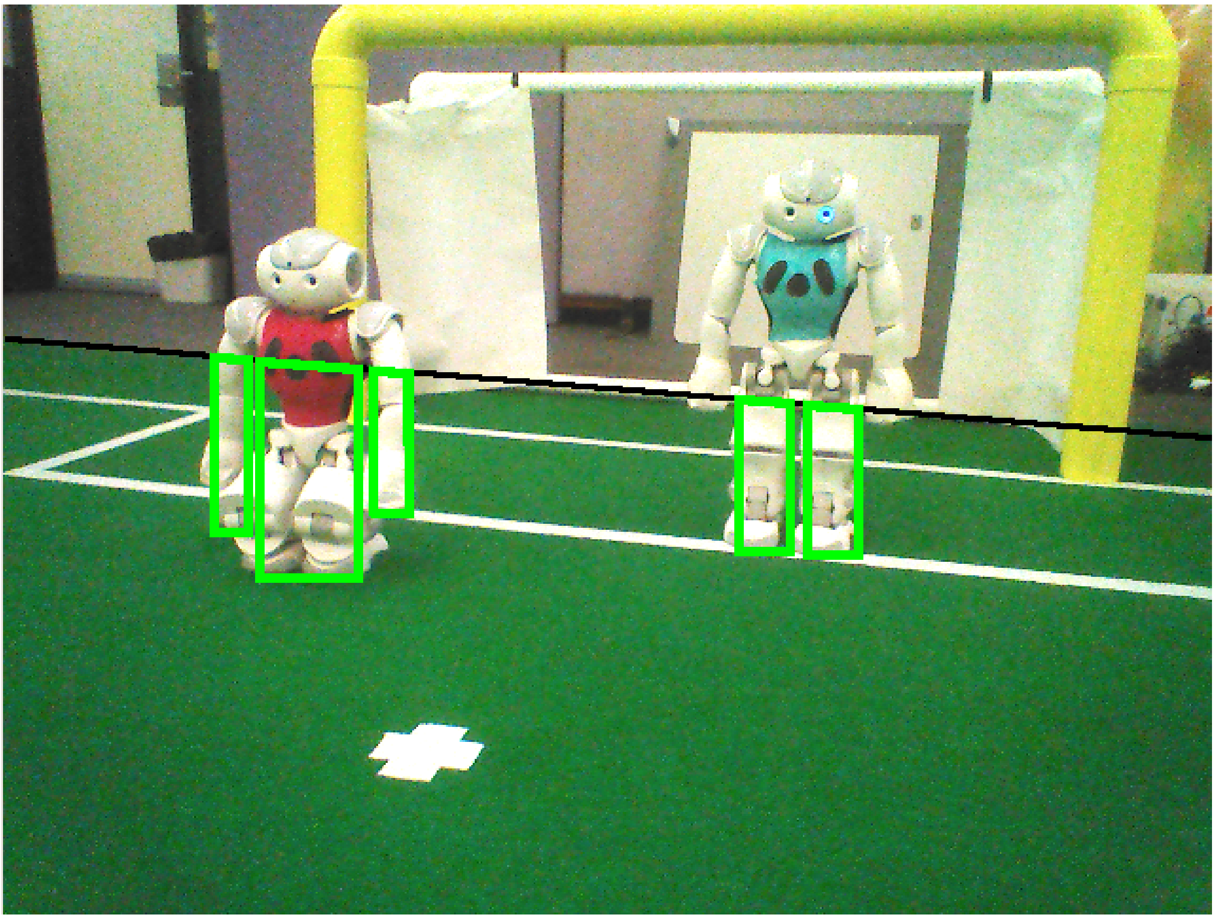

Figure 3.6: Goal for obstruction bottom detection

Implementations

This piece of code had the most changes throughout the implementation because of it's importance to the rest of the robot detection and of failed attempts at making it work effectively.

The first attempt was to look at the edge density below the obstruction line. This uses the assumption that a robot contains a lot of colour changes, compared to the field carpet, and therefore when the high rate of edge changes drops off the carpet has been reached. As can be seen in Figure 3.7, this implementation works very well for far-away obstructions as there is a clear point at the bottom of the feet where the edge density drops. However, when the robot moves closer to the obstructions the edge density becomes less and less useful as can be seen in Figure 3.8. This occurs because the closer robots do not have as many edge changes because the colour is largely consistent. This prevents the bounding box from being correctly extended to the bottom of the feet.

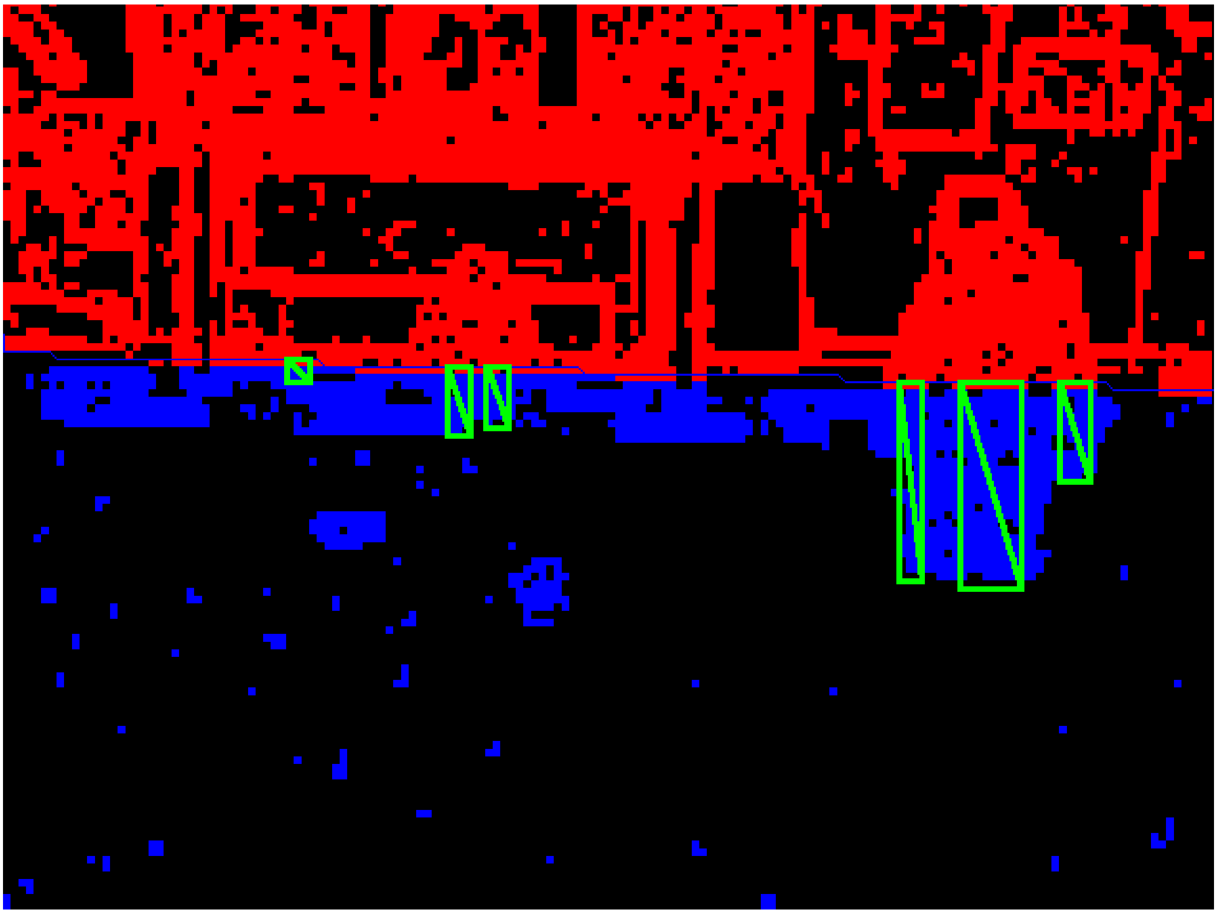

Figure 3.7: Feet detection of a far away robot using edge density

Obviously, from this experiment, edge density as currently implemented was not going to be consistent enough for all cases. An implementation that uses edge density for far-away obstacles and a different method for close-up could be used but this deterministic code is

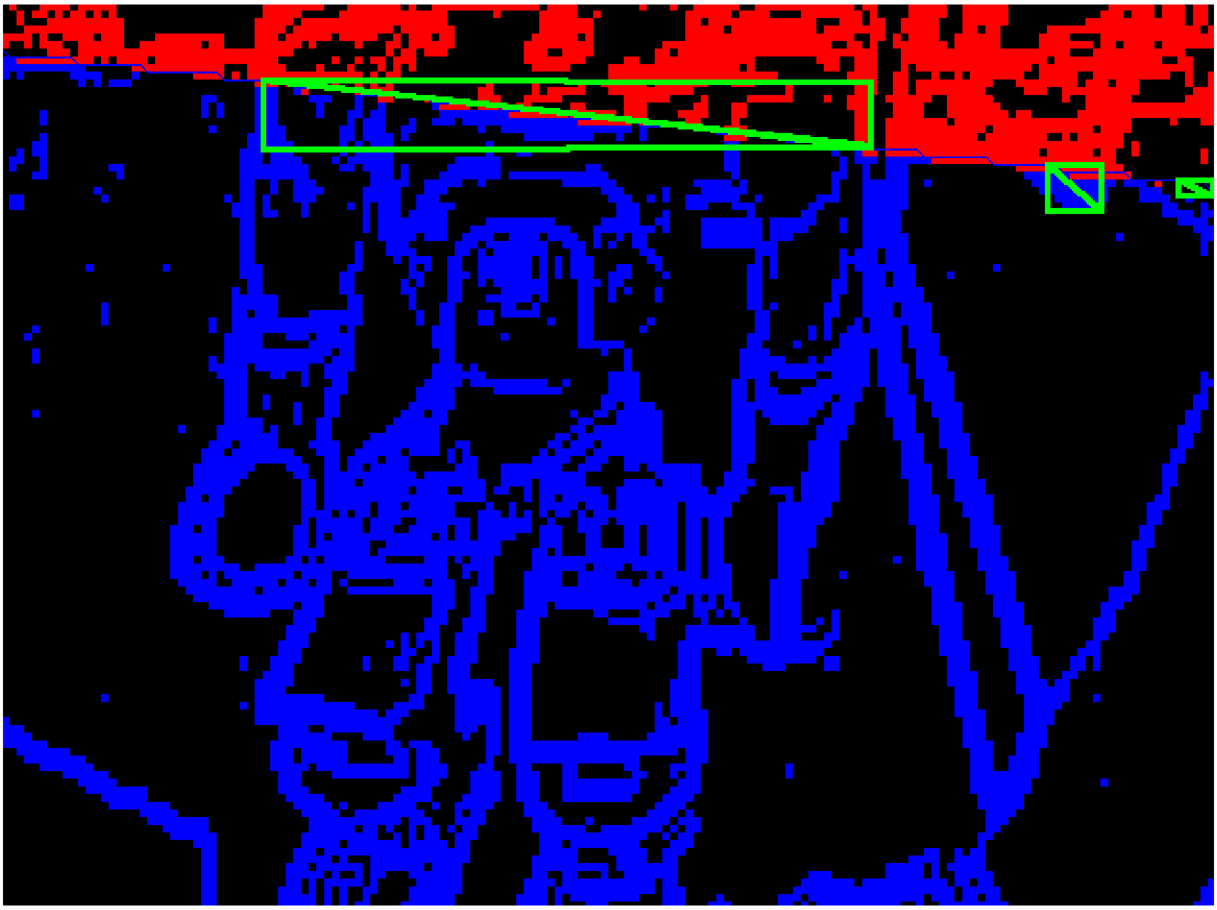

Figure 3.8: Feet detection of a close robot using edge density

undesirable. Because of this, the final feet detection algorithm used the colour to determine the bottom of the feet like the old robot detection.

As the old robot detection also used colour, the new implementation's goal was to create a better algorithm using the colour. The old bottom obstruction detection can be seen in Figure 3.9 where the red line in this image represents where it thinks the bottom of the obstruction is. It can be seen that it struggles if there is a significant amount of green on the robot. This could be caused by bad colour calibration or shadows on the robot. In the new robot detection code, this problem was to be reduced so that it could find the true bottom.

The new robot detection code got around this by applying a merging implementation similar to the obstruction detection code above. It looked down from the obstruction line and, if that row of pixels was mostly non-green, it considered it still part of the obstruction

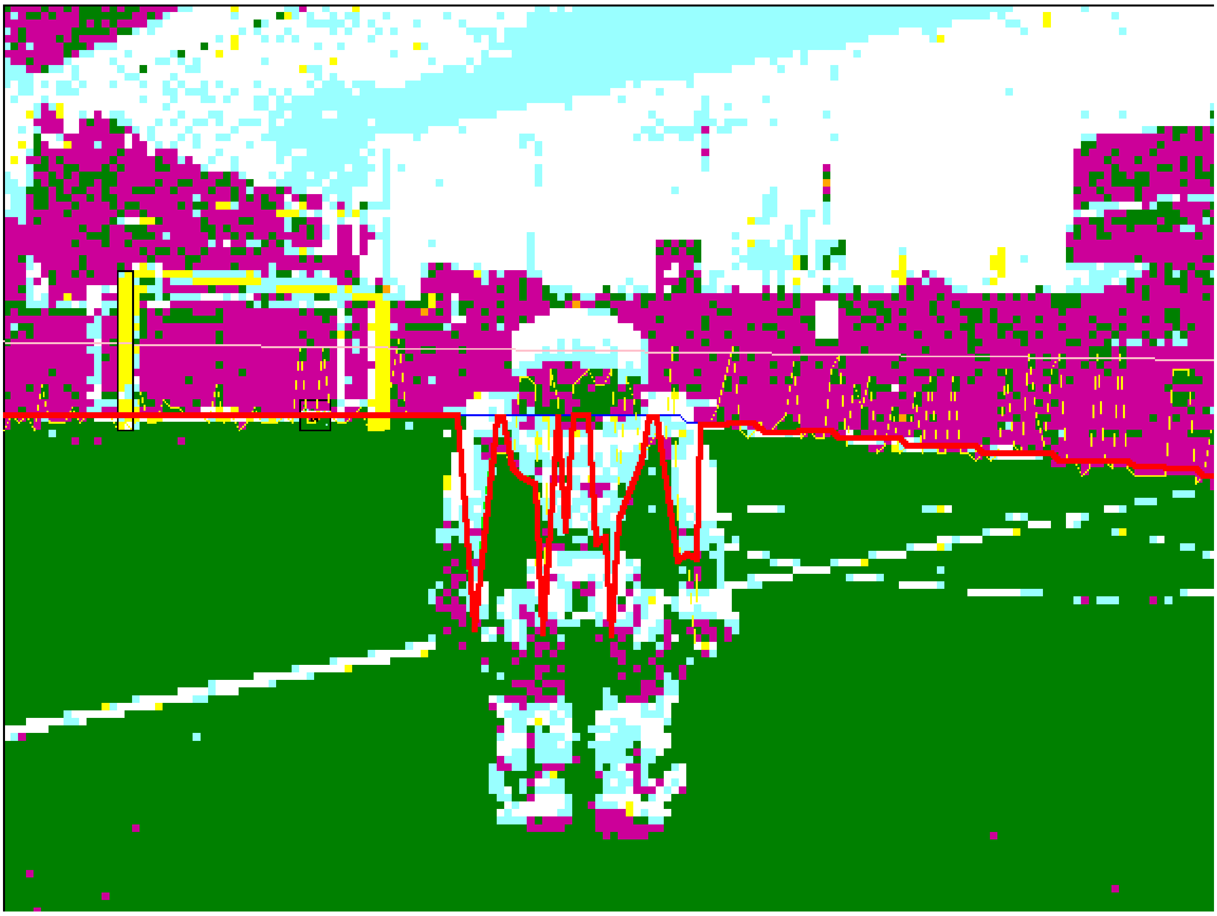

Figure 3.9: Original robot detection bottom obstruction detection

and kept looking down. If there was a significant number of consecutive rows that were green, e.g. 5, it would stop looking down. This fixed the problem caused by a bit of green on the robot stopping the algorithm from finding the true bottom, which was a problem in the old robot detection code.

Result

The bottom detection of the algorithm did quite well as can be seen in Figure 3.10.

The algorithm was further tested on a larger sample set and the data seen in Table 3.2.2 was collected.

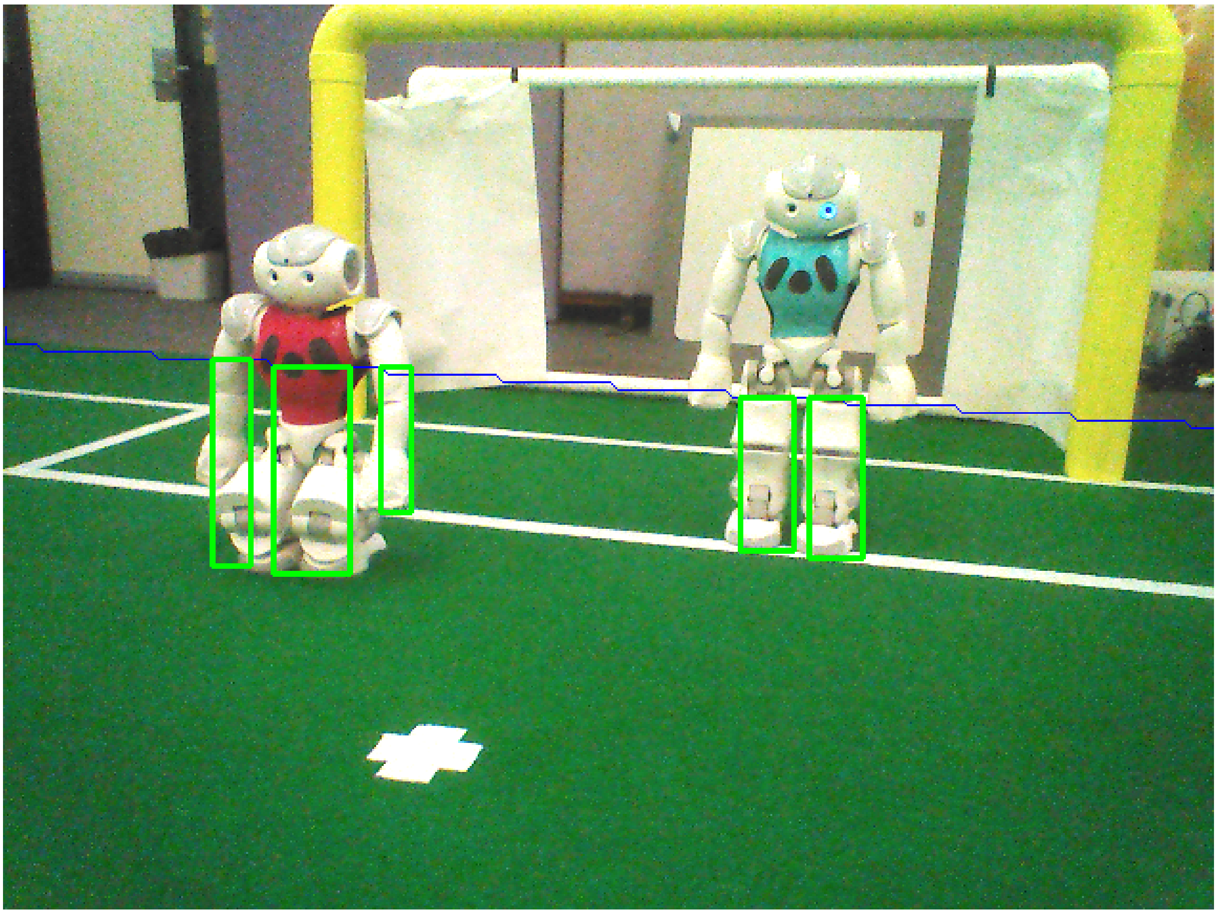

Figure 3.10: Result for obstruction detection

$$\begin{aligned} SuccessRate &= \frac{\text{Extended correctly}}{\text{total}} \\ &= \frac{131}{169} \\ &= 77.5\% \end{aligned} \tag{3.3}$$

The test set was run again and this time the number of pixels that the bottom detection was off by was recorded, e.g. 0 means perfect x means it needed to be lower or higher by x pixels.

Table 3.1: Results for obstruction bottom detection

Extended correctly	131
Extended too far	22
Extended too short	16
Total	169

Data set

0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 36, 2, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 2, 0, 0, 53, 0, 0, 0, 0, 0, 0, 54, 1, 1, 1, 45, 0, 0, 0, 48, 0, 0, 50, 0, 0, 20, 23, 19, 20, 10, 2, 15, 30, 16, 12, 20, 15, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 12, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 10, 0, 0, 0, 12, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 5, 2, 2, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

- **Count**: 172
- **Sum**: 568
- **Mean**: $\frac{568}{172} = 3.302$
- **Standard deviation**: 9.820

Analysis

From the data, it can see that the bottom detection had a success rate of only 77.5%, which is not desirable. However, the second experiment showed that, on average, the detected bottom was only 3.302 pixels off the desired position. Therefore, the margin of error in this module is not significant and small improvements would easily be able to increase the overall success rate.

3.2.3 Extending to the robot head

Goal

The next step in the robot detection was to determine the top of the robot, e.g. the head. The old robot detection did not have this functionality so adding it would allow better determination of whether the detected object was a robot in future sanity checks/machine learning. As we can see in Figure 3.11, we want to extend the bounding box created from the feet detector to the top of the head.

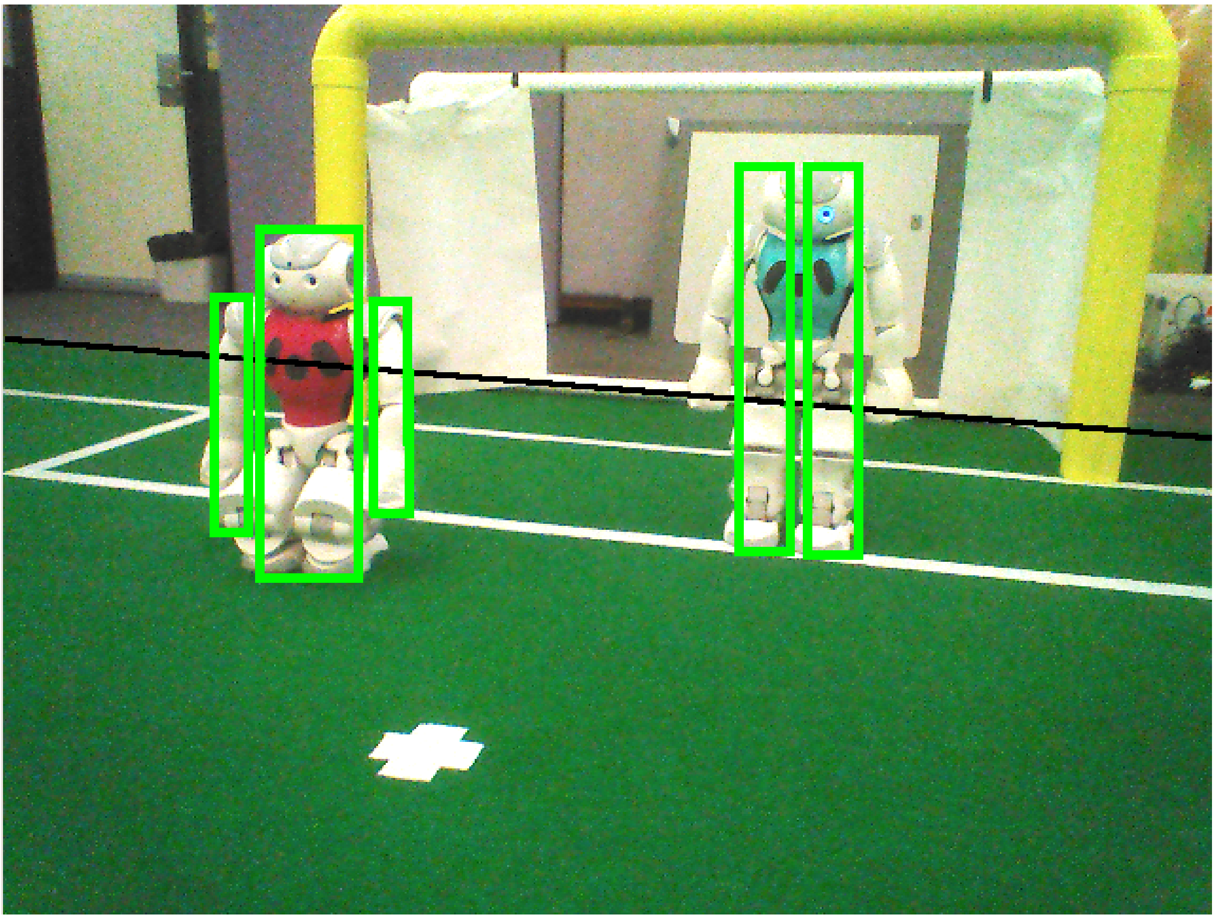

Figure 3.11: Goal for robot top estimation.

Implementation

Due to the inconsistencies of the possible colours above the field edge, a method using colours or edges would not be a reliable method to determine the top of the robot. For example, an algorithm that just uses the colour of the robots to extend up would break if the field background was white, blue or red in colour as the algorithm would not know when the robot ends and the background begins.

Another method, and the one that was eventually chosen, involved using camera information and trigonometry to calculate where the top of the robot will be in the image space. By calculating the bottom of the robot, the distance to the robot's feet is able to be calculated. This is done by using the cameraToRR code that was already residing in the code base. This code retuns a Robot Relative coordinate storing the distance and heading to the bottom of the robot.

Another fact that is known about the robot is the height of the robot. As all the robots are the same height it can be hard coded to have the robots a generic height, 500mm in this case. At this point the assumption is made that all robots are standing and not crouching. If they are crouching the box will be a bit higher then the robot, which is undesirable but does not cause any major problems.

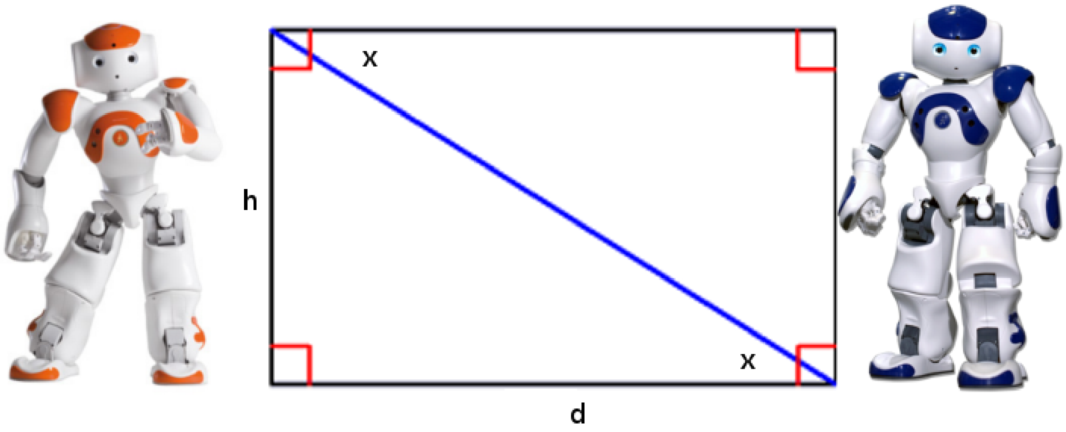

Figure 3.12: Trigonometry for calculating the angle for the seen robot

Figure 3.12 shows the trigonometry that would be used for the calculation. The angle x needs to be found as it represents the angle of the object in the camera space. This can be calculated by using trigonometry as seen in Equation 3.4.

$$x = \tan^{-1}(\frac{h}{d}) \tag{3.4}$$

As adjacent angles on an intersecting line of two parallel lines are equal the top angle seen in Figure 3.12 is the same as the bottom angle. When this is calculated, an estimated angle for the object in the camera space has been obtained. Given the vertical field of view (angle that camera sees) and the height in pixels of the image, the height in pixels of a sub-angle in the field of view can be calculated. The simplified code can be seen in Listing 3.1.

In the current NAO the vertical field of view is 60.9°and the vertical height of the image in pixels is 960. The number of pixels each degree represents is calculated by dividing the pixels by the field of view degrees, which in this case is 1°= 15.7pixels. Therefore, if the angle of the object is 20°the height of the object can be deduced as 315 pixels above the feet of the robot.

Listing 3.1: Simplified code to calculate height of robot in pixels

```
function getPixelHeight(degrees) {
    field_of_view_in_degrees = 60.9
    image_rows = 960

    pixels_per_degree = image_rows / field_of_view_in_degrees

    pixel_height = degrees * pixels_per_degree
    return pixel_height
}
```

Result

Because of the assumption that the robot is standing up the calculation does not do as well as expected when the robot is crouched, however, it does relatively well with standing robots as seen in Figure 3.13.

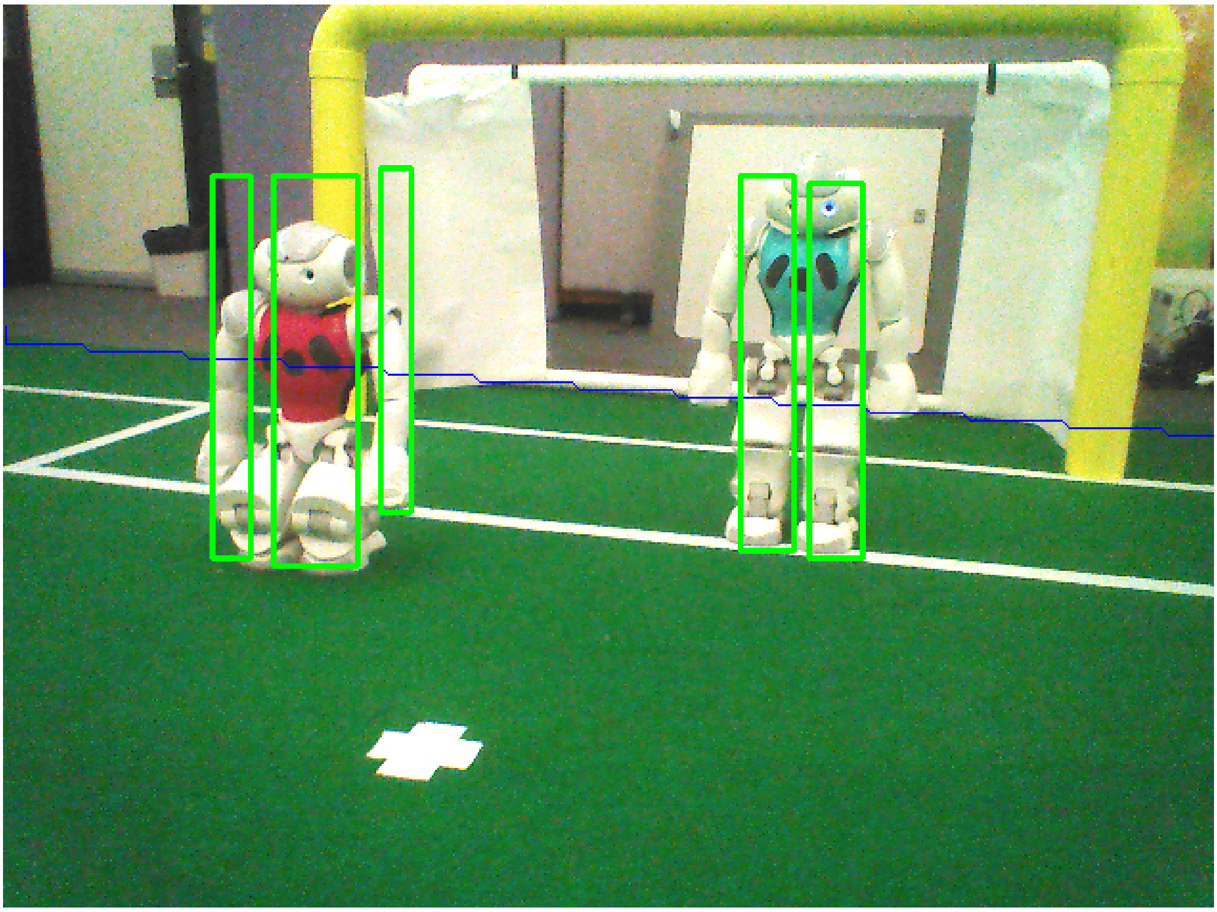

Figure 3.13: Result for top of obstruction estimation.

This was tested on the sample set and the results are shown in Table 3.2.5.

Analysis

As can be seen in Table 3.2.5, the height estimation is estimating the height to be too high. This is caused by the crouching of the robot, as well as, detected robots at the arms being extended too far as it thinks they are also feet of a robot. In both of these cases, the consequence of over extending as detrimental as they would be. The robots are

Table 3.2: Height estimation results

Extended correctly	64
Extended too far	36
Extended too short	3
Total	103

not often crouching, and over extending of the arms is not such a problem as they are later merged into the body. This extra height in the arms does not matter in the rest of the code. However, improving these aspects of the height estimation would improve the algorithm overall and produce a semantically better height estimation.

3.2.4 Applying sonar information

Goal

As well as using the visual information for the robot, the sonar data could be greatly utilised to detect robots. For example, if a robot is seen a metre in front and the sonar information says there is also an object a metre in front, there is a big indication that there is an obstacle a metre in front. The goal of this step is to read the sonar information and mark any obstructions that the sonar would likely apply to.

Implementation

The sonar information received by the robot comes from the left and right sonar sensors. This information is stored in three lists, left, right and middle with values for the objects that it has detected. For example, the left sensor could have readings [120, 360] indicating it sensed objects 120mm and 360mm away. Each of the sensor lists represent a horizontal range in front of the robot. Currently, the left sensor has sonar information for the range -10°to 70°, middle -20°to 20°and right 10°to -70°. The detected robots heading is used to determine which sonar reading should be looked at.

For all the available sensor information, the closest sensor value to the visual distance is used. For example, if the visual distance is 700mm and the sensor information is 670mm and 1000mm, the obstacle would be assigned a sensor reading of 670mm.

This sensor information for the obstacle will be used in the Bayesian machine learning as a feature. It is not required but makes for better detection.

3.2.5 Bayesian Machine Learning analysis

Introduction

Naive Bayesian machine learning is an algorithm to determine if a given sample is either one state or another, in this case a robot or not. The algorithm uses a large training set with each sample marked as being a robot or not, as well as including the feature information for it. For each of the features in the sample (e.g. percentage of white) a probability that the feature indicates it is a robot or not needs to be calculated. For each of these features, the probabilities are multiplied together to get a final probability that it is a robot or not. This probability is multiplied with the prior probability that it is a robot or not, and if the probability that it is a robot is higher than not, it is assigned as a robot and vice versa.

For explanation purposes, there is a training set with 66% examples being a robot and 33% not robots and three features A, B and C. Each example to be tested would calculate a value for the features X, Y and Z respectively and pass it into the Bayesian machine learning algorithm. This would then output probabilities for these values that it is a robot or not, which can be seen below:

- **Prior**: probability robot = 66%, probability not a robot = 33%
- **A** : probability robot = 45%, probability not a robot = 60%
- **B** : probability robot = 50%, probability not a robot = 40%
- **C** : probability robot = 30%, probability not a robot = 80%

The following calculation can now be done to find out the probabilities:

Note: for stopping the problem where multiplying the probabilities will result in very small numbers, which on a computer may round 0.00001 to 0, a summation of the log of the probabilities will be used instead.

$$\begin{aligned}\text{Probability robot} =& Prior(Robot) * A(Robot) * B(Robot) * C(Robot) \\ &= 0.66 * 0.45 * 0.5 * 0.3 \\ &= log(0.66) + log(0.45) + log(0.5) + log(0.3) \\ &= -1.3511\end{aligned} \tag{3.5}$$

$$\begin{aligned}\text{Probability not a robot} =& Prior(NotRobot) * A(NotRobot) * B(NotRobot) * C(NotRobot) \\ &= 0.33 * 0.6 * 0.4 * 0.8 \\ &= log(0.33) + log(0.6) + log(0.4) + log(0.8) \\ &= 1.1982\end{aligned} \tag{3.6}$$

Therefore for this example $-1.1982 > -1.3511$, so it would be assigned as not being a robot.

Why Bayesian?

In the 2011 robot detection code[1], a decision tree machine learning algorithm was used for the analysis of the robot. A snippet of the decision tree can be seen in Figure 3.14.

While this decision tree may work well, it does have the restriction that it sets an example as not a robot if one of those conditions is not true, even if all the other conditions are true. The Bayesian machine learning has an advantage over the decision tree as it considers all of the features and, at the end, it calculates a percentage probability that it is a robot or not.

Goal

The goal of the Bayesian machine learning algorithm is to remove the false positives detected by the robot detection. Cases where it has detected a referees leg, for example, should be removed or if it has thought that a line was a robot should also be removed. It should also keep all the true positives.

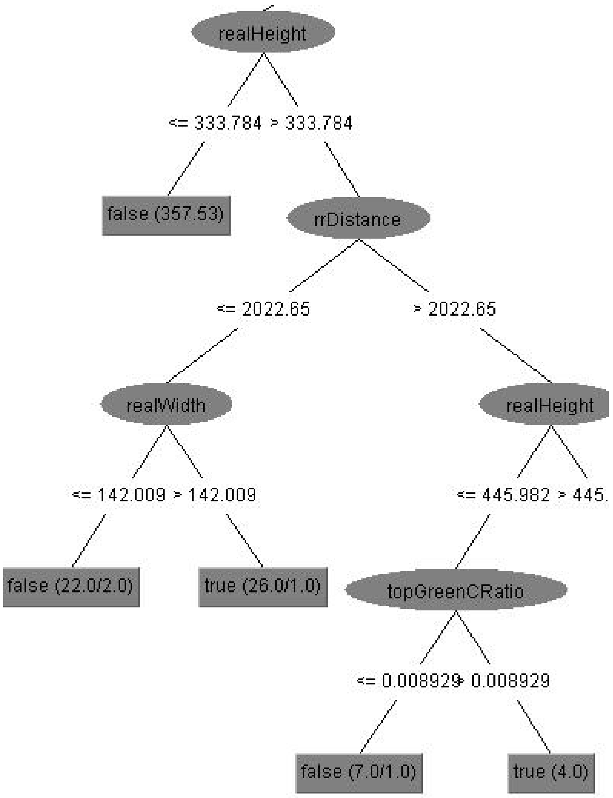

Figure 3.14: Snippet of the decision tree used in the 2011 robot detection code[1]

Taking in training data

To get the training set for the Bayesian machine learning, an example training set data was recorded from the robot. This training set needs to include all the different cases for viewing the robot. For example, it should include images of the robot directly in front, as well on the side, close and far, etc. Some of these frames can be seen in Figure 3.15

From these sample frames, the code described above is run on the frame, which will result in possible robots to be detected as can be seen in Figure 3.16. From here, the program outputs the following from the robot:

- **Gradient**: the boxes rise over run. This is used because false robots are often very tall or very flat and thus should negatively impact the machine learning.
- **Percentage of white**: given the pixels in the box, what percentage are white. Often a false positive, such as detecting the referees leg will have very little white.

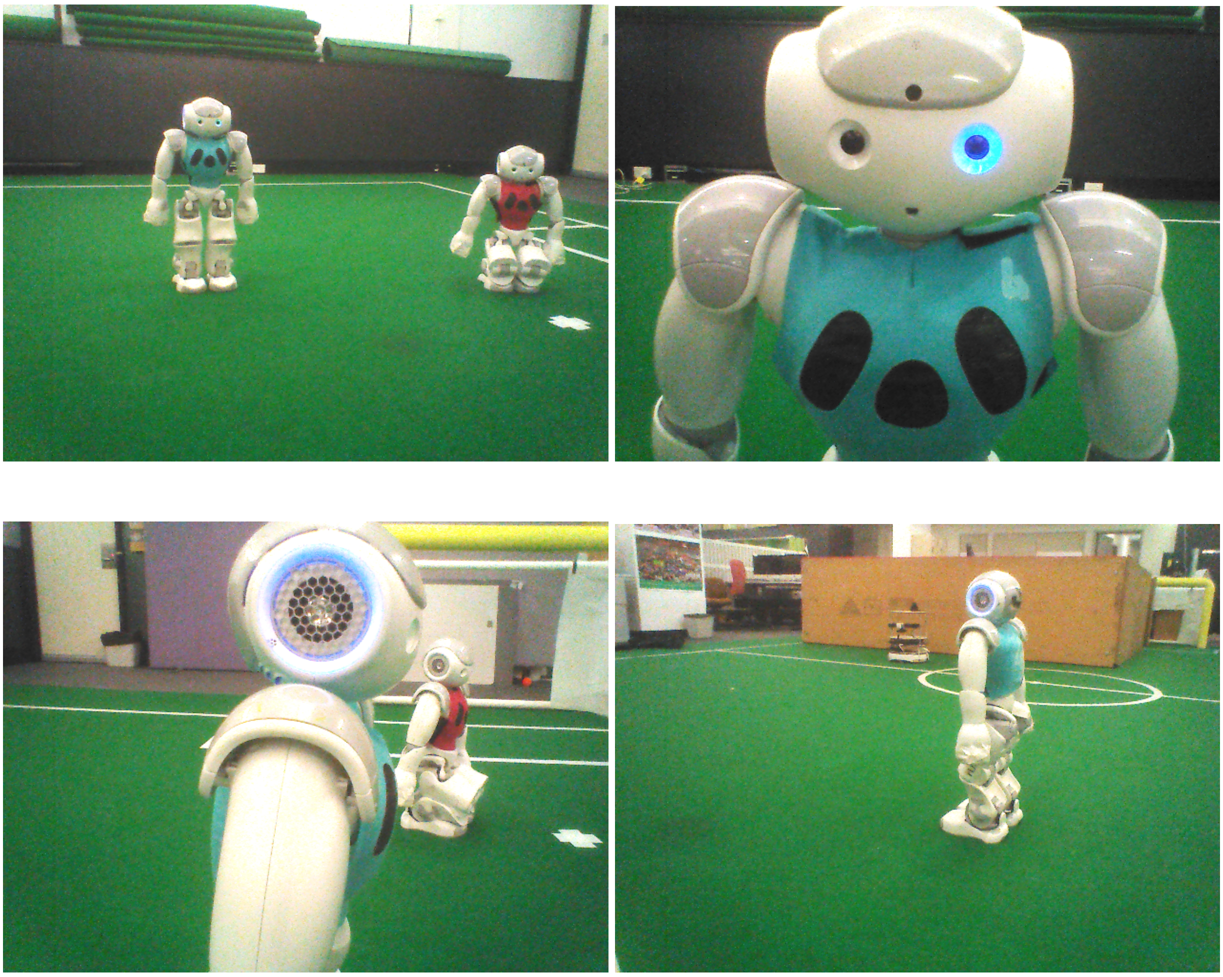

Figure 3.15: Some sample frames for Bayesian machine learning

- **Percentage of red**: given the pixels in the box, what percentage are red. This is used to positively impact the machine learning score if it sees the jersey.
- **Percentage of blue**: given the pixels in the box, what percentage are blue. This is used to positively impact the machine learning score if it sees the jersey.
- **Percentage sonar difference**: given a visual distance of x and a sonar distance of y, the percentage sonar difference is: $\frac{|x-y|}{x}$, e.g. the difference between the sonar and visual distance over the visual distance.

Table 3.3 includes some of the training data that is used in the algorithm. *Note: information about whether it is a blue, red or arm is recorded but ignored and all of these three cases are considered true. It has been marked this way in case this wants to be included in the code in the future. This could be implemented by calculating the proba-*

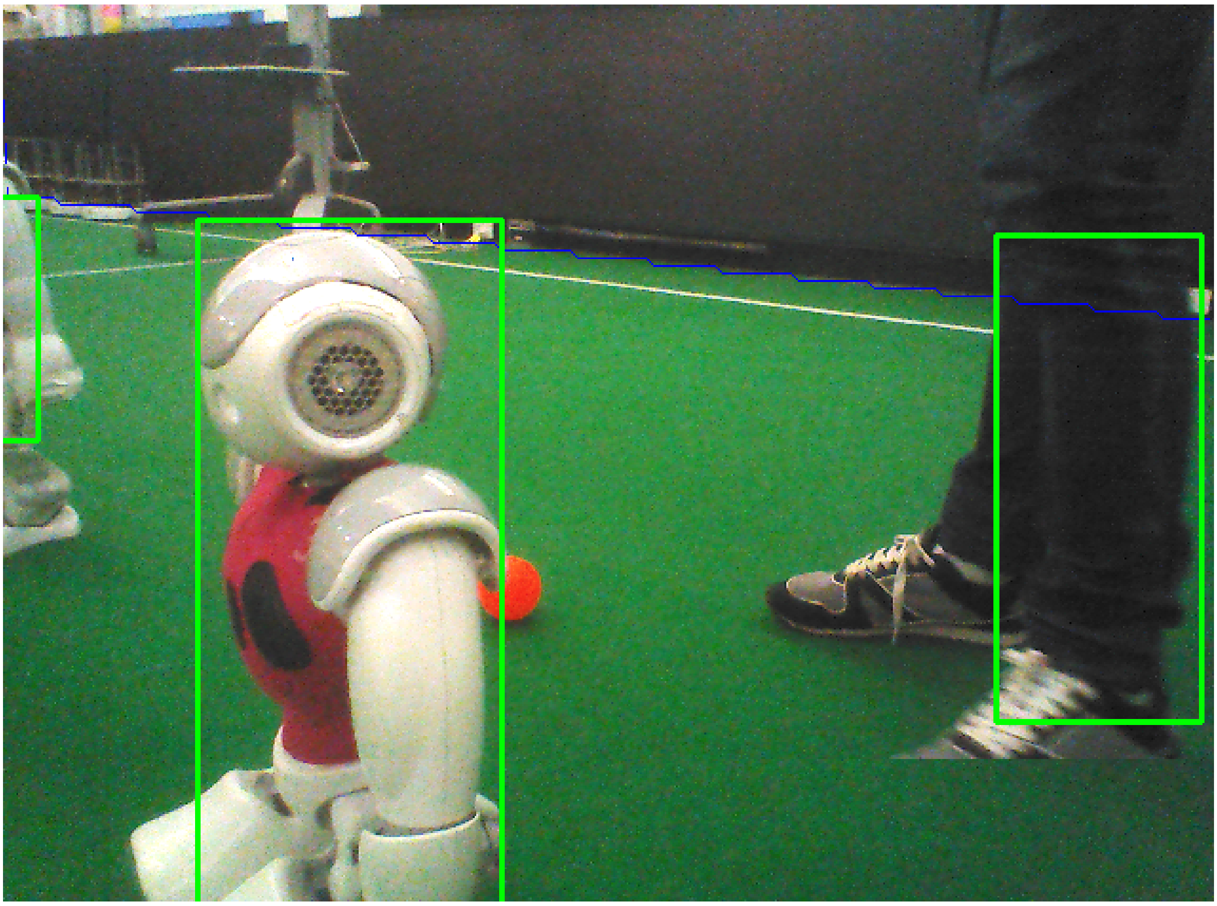

Figure 3.16: Frame with detection code run on it

bility that it is an arm, body(red or blue) or not a robot, instead of just being a robot or not.

Gradient	%white	%red	%blue	%sonarDifference	false/red/blue/arm
3.5385	0.5284	0.0351	0.1371	0.0586	blue
11.0000	0.4141	0.0404	0.0505	0.3418	arm
2.0909	0.2490	0.1146	0.0316	0.5434	arm
3.1667	0.0351	0.4912	0.0088	0.6152	false
1.6667	0.1625	0.3000	0.0125	0.6072	false
9.3333	0.0952	0.0714	0.0595	0.3269	false
3.1429	0.4156	0.1494	0.0130	0.4766	red

Table 3.3: Example training set data for Bayesian machine learning

Feature probability calculation

This section describes how the probability that it is a robot is calculated given a value for a feature. For example, if the gradient of the robot is 5, this must be able to return a probability that this is a good gradient for a robot or not. For these features, a bucketing system will be used as it is simple to calculate and is not CPU intensive. This works by taking the feature value, then determining what bucket it is within, and calculating the probability that it is a robot if it is in that bucket.

The following is an example bucketing system for a feature. In this example, there is a feature with 3 buckets, A, B and C, and the probability that is a robot or not given a value in the bucket is wanting to be calculated. This is done by going through the training set and incrementing the number of positive robots that are in each bucket, with the same being done for the negative. For example, there may be 100 sample set and 60 are positive examples and 40 are negative the following values could be recorded as seen in Table 3.4.

Feature Bucket	Is a robot	Is not a robot
A	30	10
B	20	10
C	10	20
Total	60	40

Table 3.4: Example bucket totals for a Bayesian feature

We can use these totals to calculate a percentage probability that it is a robot by dividing the total for that feature by the total number of robots. This would give the following, as shown in Table 3.5.

The buckets for each of the features used in the robot detection Bayesian learning is seen in Figure 3.17

Feature Bucket	Is a robot	Is not a robot
A	50%	25%
B	33%	25%
C	16%	50%

Table 3.5: Example bucket percentages for a Bayesian feature

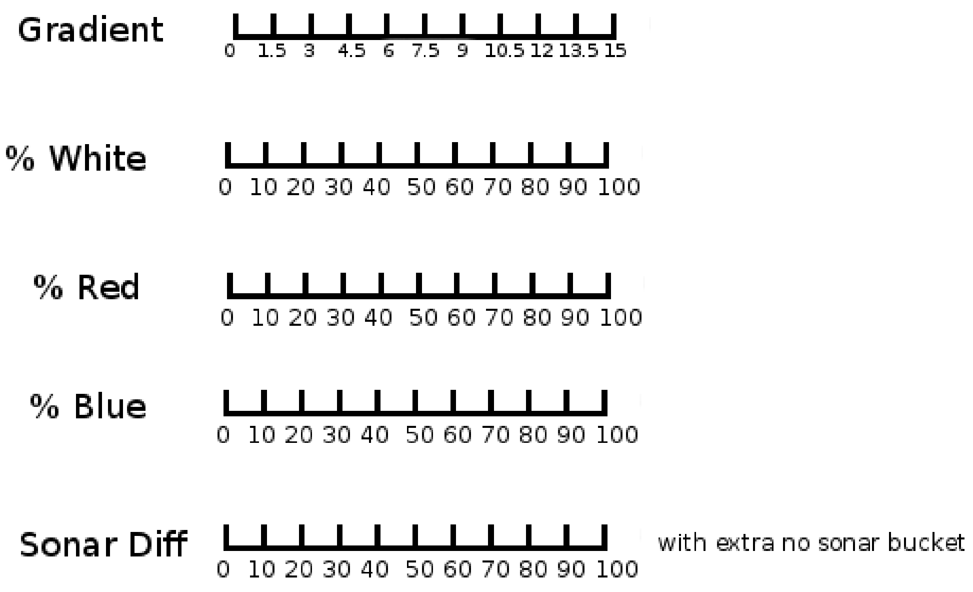

Figure 3.17: Buckets used for each feature in the Bayesian ML algorithm

Result

Figure 3.18 shows an example application of the Bayesian machine learning algorithm. As can be seen, it was able to distinguish between the robots and the any false positives that that was previously detected by the robot detection algorithm.

$$\begin{aligned} Sensitivity &= \frac{\text{True positives}}{\text{True positives} + \text{False Negatives}} \\ &= \frac{141}{141 + 26} \\ &= 84.4\% \end{aligned} \tag{3.7}$$

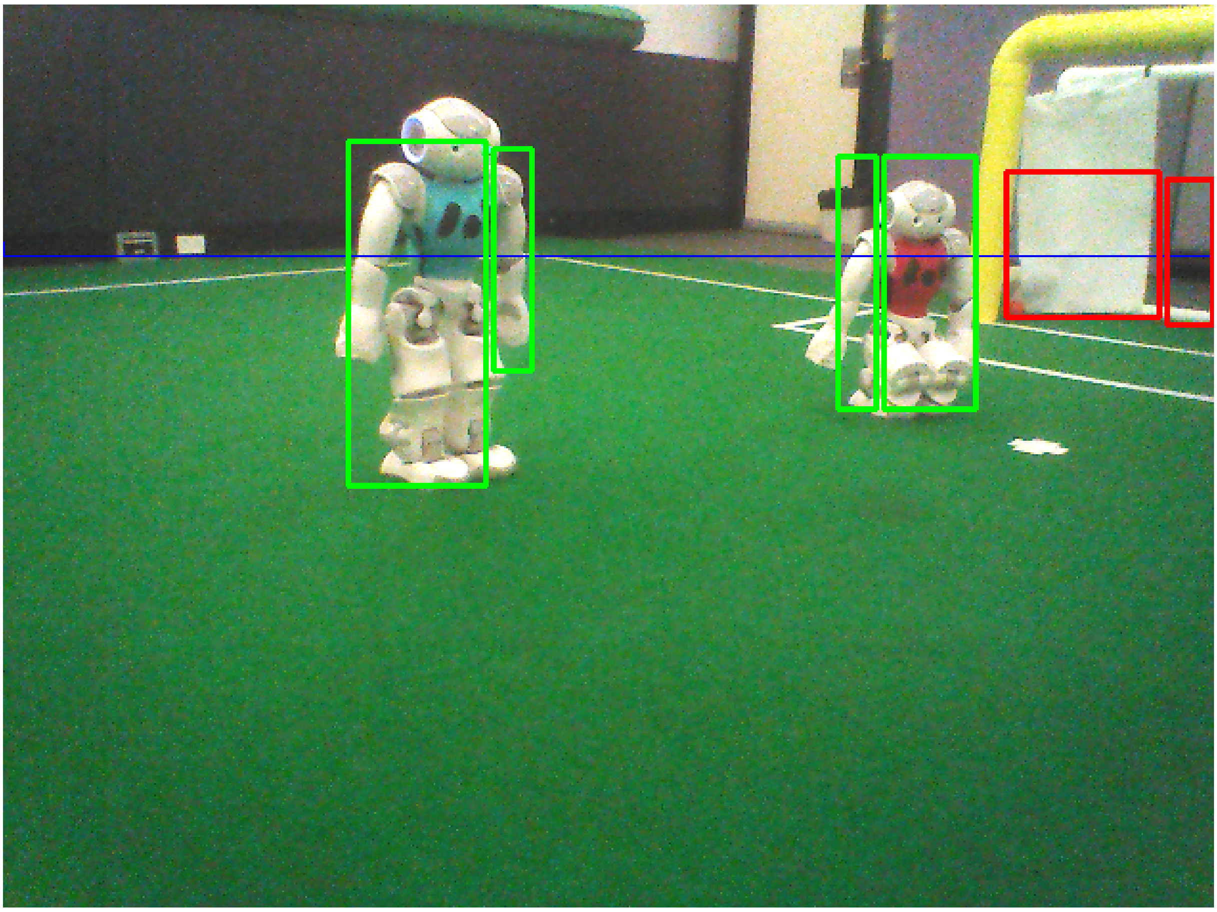

Figure 3.18: Example Bayesian application; green is a robot, red is not a robot

	Is a robot	Is not a robot	
Detected as robot	141	16	157
Not detected as robot	26	34	60
	167	50	217

$$
\begin{aligned}
Specificity &= \frac{\text{True negative}}{\text{True negative} + \text{False positive}} \\
&= \frac{34}{34+16} \\
&= 68\%
\end{aligned}
\tag{3.8}
$$

Analysis

As can be seen with the 84.4% sensitivity, it was able to detect a high proportion of the robots correctly. However, it did not do as well at removing the negative cases, with only a 68% rate. Some of the bad cases can be seen Figure 3.19 where a bad camera calibration for the lower area of the robot has caused bad detection and analysis. This indicates the need for a good colour calibration for detecting the robots close up.

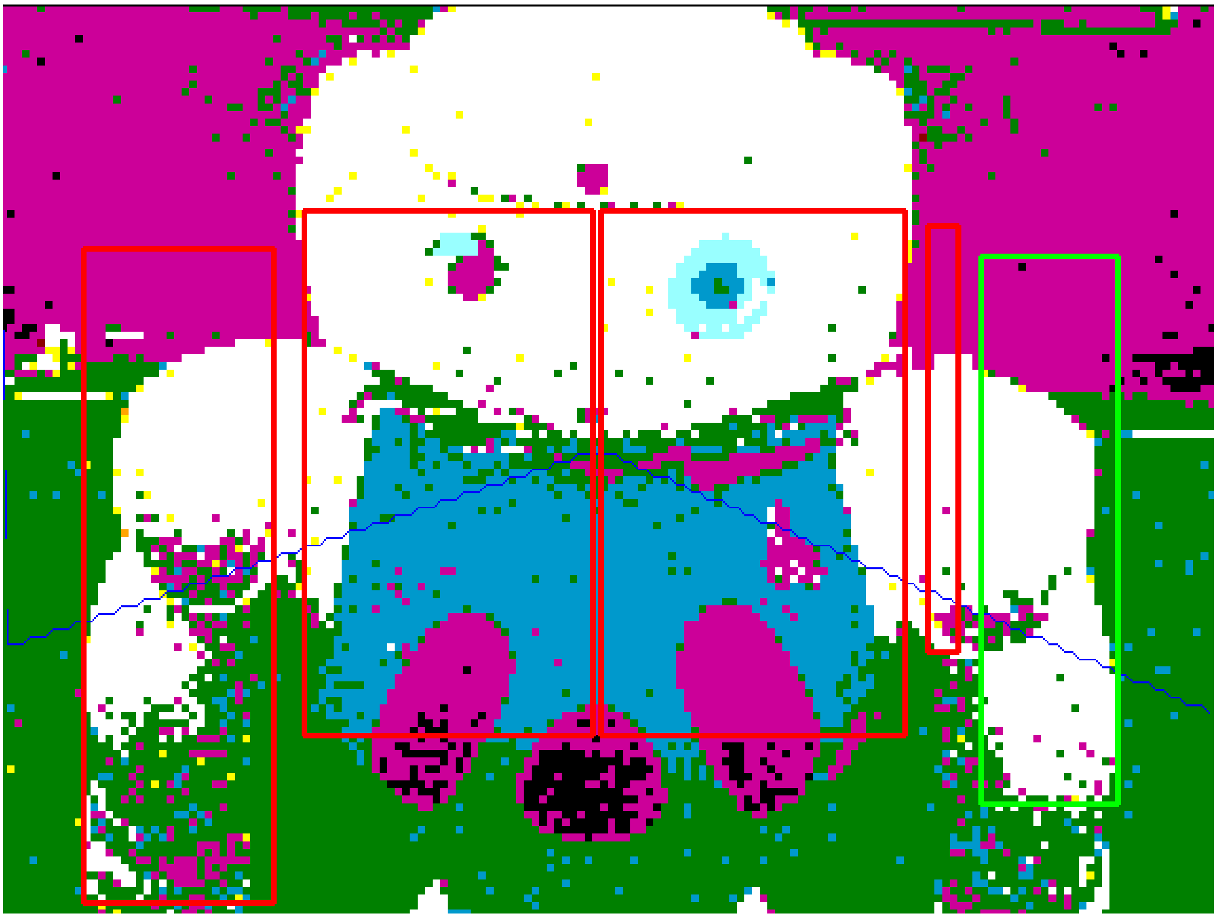

Figure 3.19: Example bad Bayesian machine learning

3.2.6 Team detection using jerseys

Goal

The goal of this step is to view the possible robots and to determine what team it is (if any, e.g. an arm shouldn't have a team and should be considered unknown). This is shown in Figure 3.20 where each robot's team has been detected.

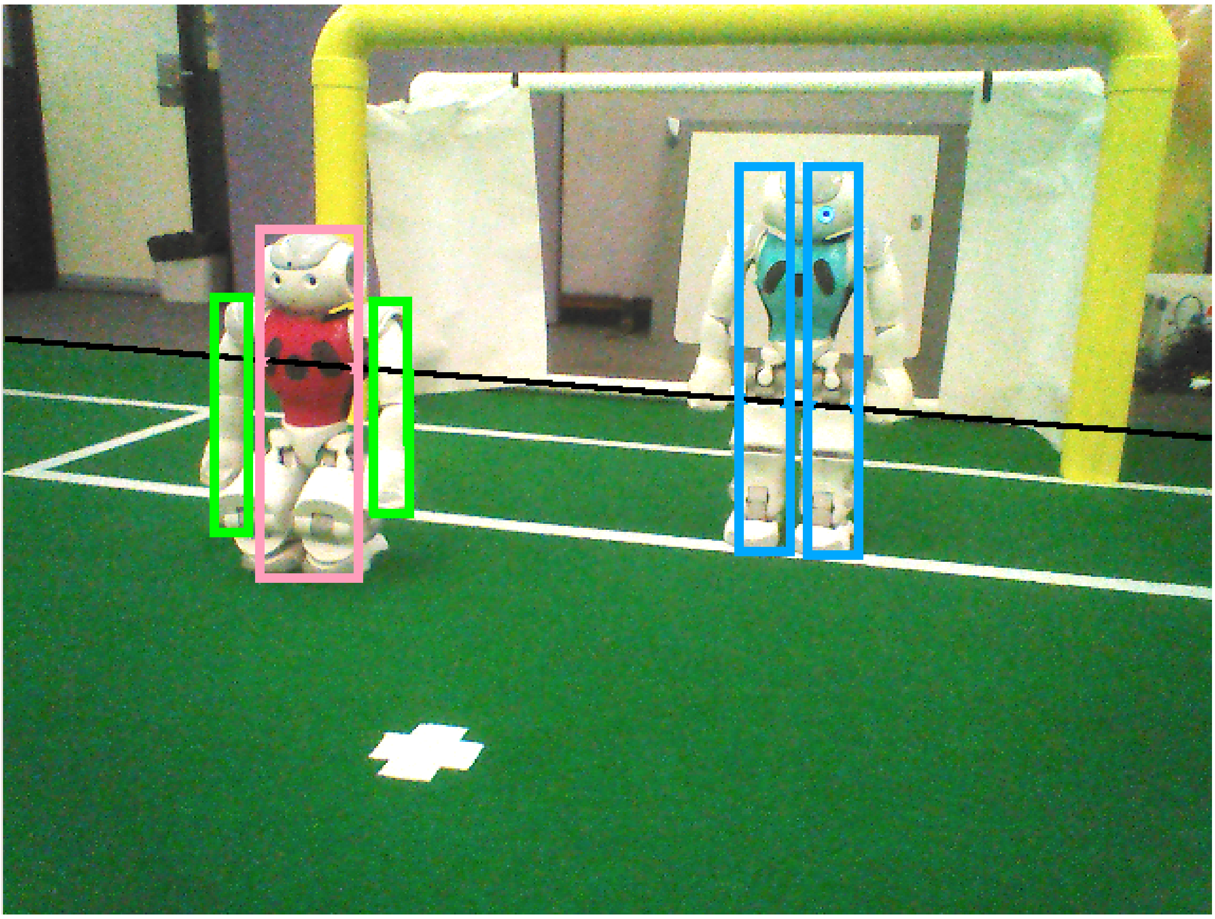

Figure 3.20: Jersey detection goal

Failed Implementations

To determine the teams for the possible robots, the algorithm needs to try and find the jersey on the robot and detect whether it is blue or red. Over the course of the implementation, many different types of detection were used.

The first method was starting at the bottom and looking up until there is a significant row of jersey colours. This worked for a few cases but did not work when the potential robot is facing the observing robot and the arms have been included. As the jersey width is thin compared to the whole width of the robot's torso and arms, it did not detect the jersey particularily well in these cases.

The other method was guessing where the jersey would be and counting the number of jersey pixels. If there was a significant number of a certain colour then it would assign it that colour. The problem with this strategy was that the estimated jersey position could be off, therefore failing to detect the jersey. Also, if the colour calibration resulted in stray pixels being observed on the robot and, if the number of stray pixels was considerable, it would falsely assign that jersey to the robot.

Final Implementation

The final implementation for the side detection used a group based detection algorithm over the entire bounding box. It would look over each pixel and, if it was a jersey colour, it would look around that pixel. If there was a significant number of pixels, and the same jersey colour around it, it would indicate that the pixel was a possible jersey. This can be seen in Figure 3.21 where the pixel is being assessed to determine whether it is a jersey pixel. Because there is a significant number of similar jersey colours around it, it would be considered a jersey pixel.

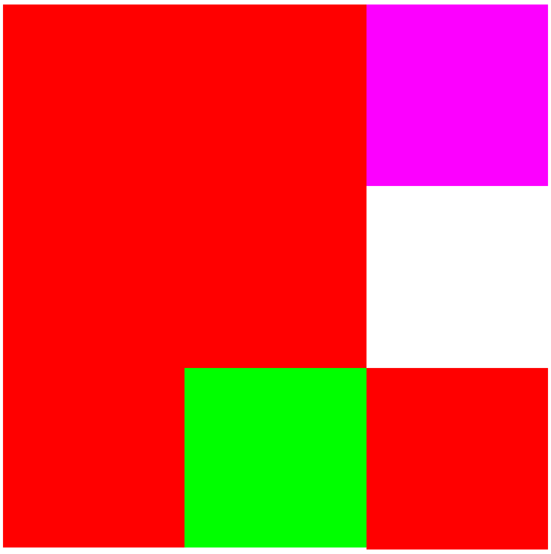

Figure 3.21: Pixel group in jersey detection

Using this pixel group detection, false detections can be removed when there is a stray jersey colour. Also, by looking over the entire sample space, the chance of missing the jersey if it is actually there will be reduced.

	Has a team	Does not have a team	
Detected team	72	3	75
Did not detect team	10	65	75
	82	13	150

Result

The jersey detection did relatively well in this sample frame as can be seen in Figure 3.22.

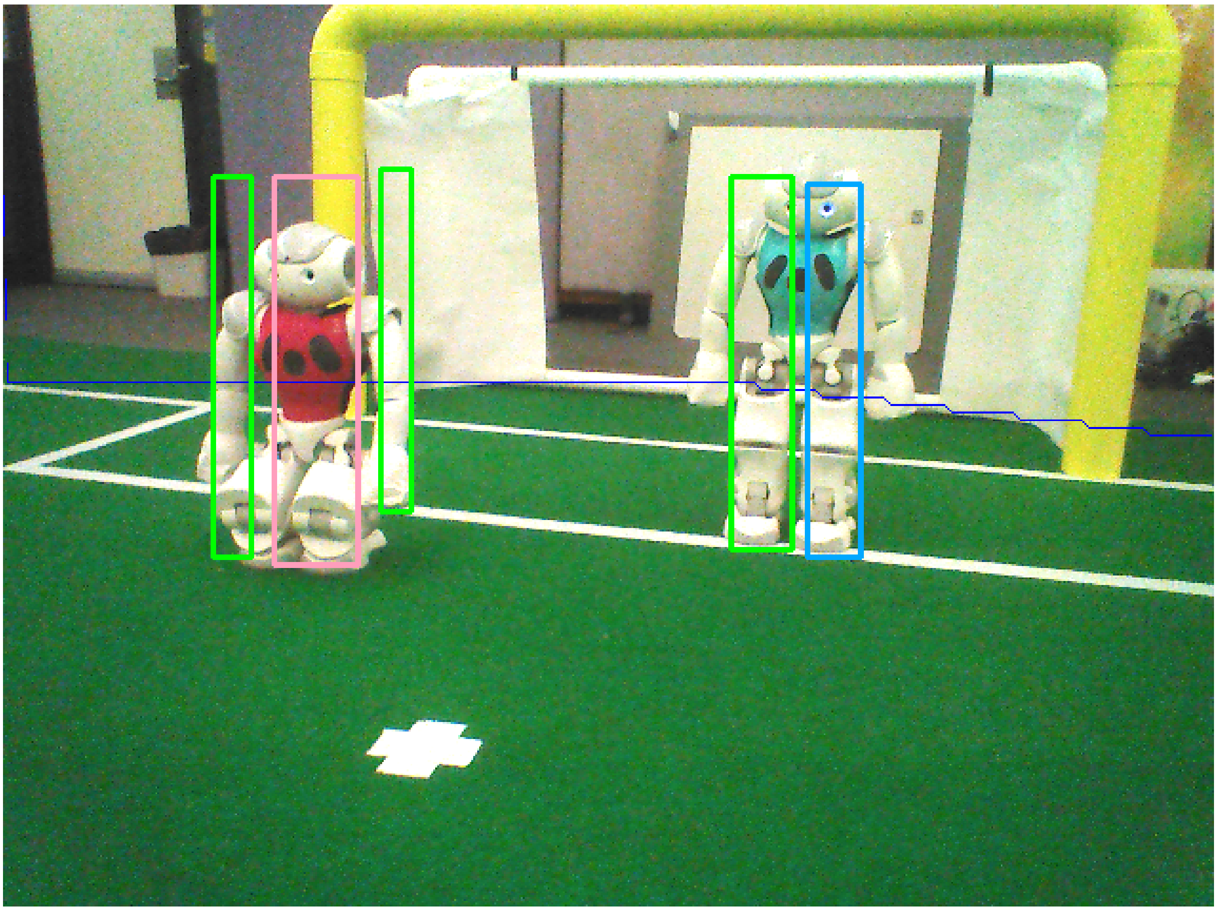

Figure 3.22: Result for jersey detection

When completing the jersey detection for all the sample set the following data was collected, as seen in Table 3.2.6.

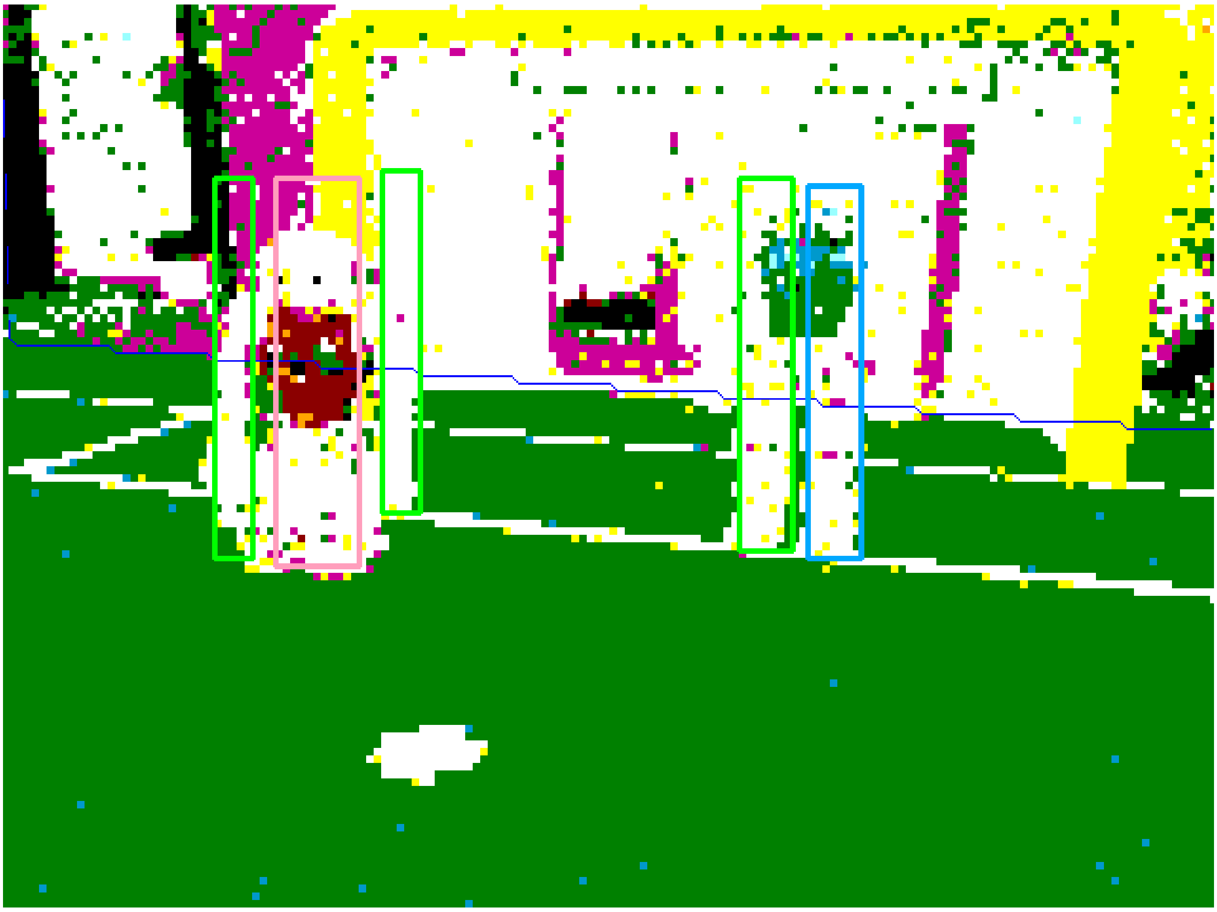

Figure 3.23: Result for jersey detection showing saliency image

$$\begin{aligned} Sensitivity &= \frac{\text{True positives}}{\text{True positives} + \text{False Negatives}} \\ &= \frac{72}{72+10} \\ &= 87.8\% \end{aligned} \tag{3.9}$$

$$\begin{aligned} Specificity &= \frac{\text{True negative}}{\text{True negative} + \text{False positive}} \\ &= \frac{65}{65+3} \\ &= 95.5\% \end{aligned} \tag{3.10}$$

Analysis

As can be seen in Table 3.2.6, the jersey detection does well in detecting the jerseys, as well as detecting when there is not a jersey. This is shown in the Sensitivity (87.8%) and specificity (95.5%) calculations. This part is heavily colour-dependent, so by looking at

Figure 3.23 it can be seen that a bad calibration will result in false negatives. From the image, it can be seen that the blue colour in the jersey is not being detected particularily well and instead it thinks it is green, thereby resulting in poor jersey detection for blue jerseys.

3.2.7 Possible robot merging

Goal

The goal of this phase is to take all of the robots and merge any groups that are close together or expected to be the same robot. For example, if there are two arms around a body it is desirable to merge this into a single robot. However, if there are two distinct robots together they should not be merged. This could happen if there are two robots close together or a distant robot with the same heading.

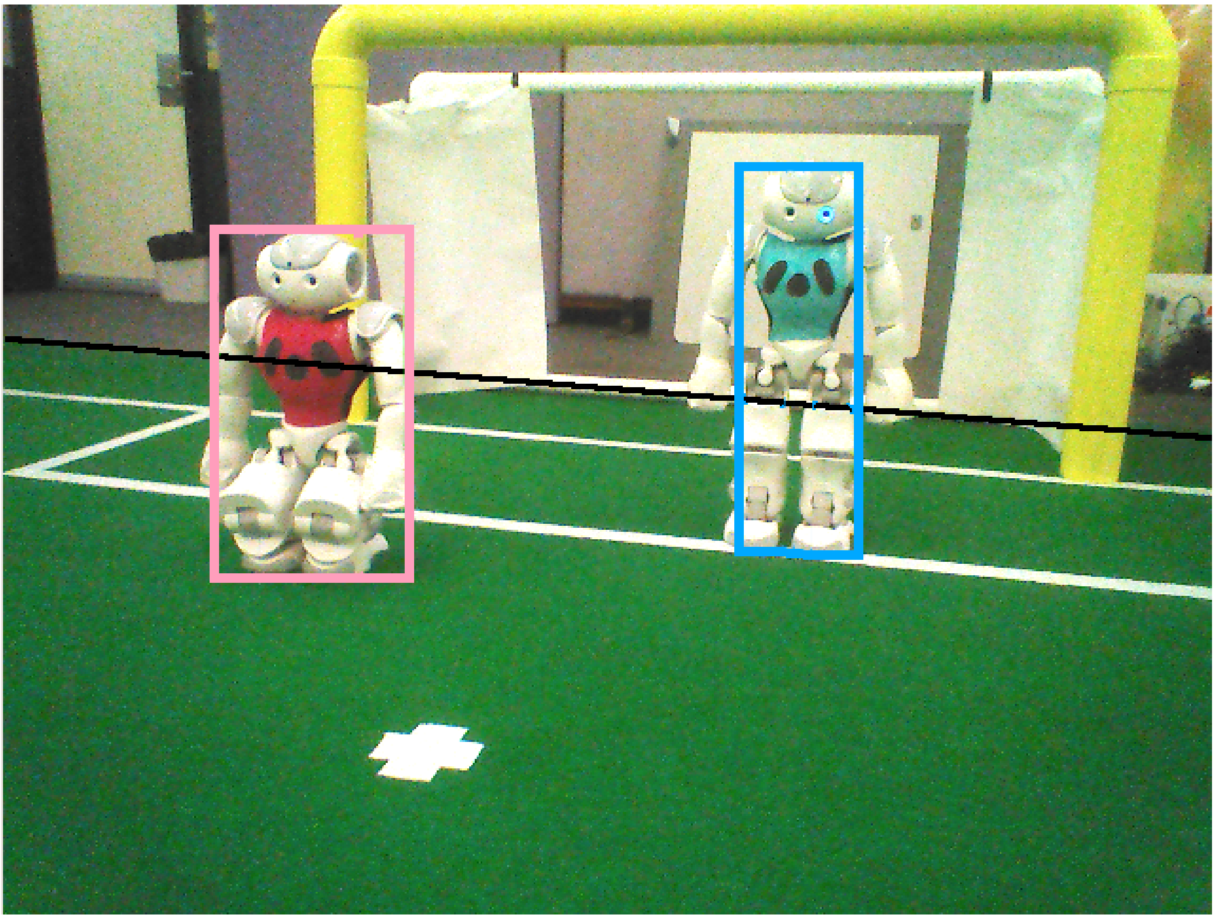

Figure 3.24: Robot merging goal

Implementation

This algorithm can be split into two sections, the first determining whether two possible robots can be merged while the second is the logic that decides when the possible robots can be merged.

Determining whether two robots can be merged is based on a relatively simple heuristic as can be seen in Listing 3.2

Listing 3.2: Simplified code to determine whether robots are merge-able

```
bool canMergeRobots(robot1, robot2) {
    bool similarTypes = areRobotsSimilarTypes(robot1, robot2);
    bool closeTogether = areRobotsCloseTogether(robot1, robot2);
    bool feetSimilarHeight = isBottomBelowHalfway(robot1, robot2);

    return similarTypes && closeTogether && feetSimilarHeight;
}
```

When determining whether two robots are similar types they can be merged if they are both the same team, or if one is unknown and the other a team, or if both are unknown. This stops two possible robots with different teams being merged together.

Two robots are considered close if the gap between them is less then the width of both the robots. The only reason for using this logic was a desire to reduce magic numbers in the code and to allow it to scale for close and far away robots. For example, a gap width of 5 pixels may work far away but in a close-up situation would not work well.

The more complicated test was the halfway check. This aimed to reduce the far away robots being merged into close robots. Often there would be an instance when where there was a robot on screen and a robot in the background. To stop these two cases from being merged together, a test was made to ensure that two possible robots must be at least halfway down each other to be merged. This can be seen in Figure 3.25 where the halfway lines are marked. If both robot's feet are below the other's halfway line, it can be merged. In this example, the far away robot is not below the half way line of the close one and therefore would not pass this test.

The next part is determining when a merge should be undertaken if it meets the following criteria. The heuristic can be seen in Listing 3.3. While the code may look complicated

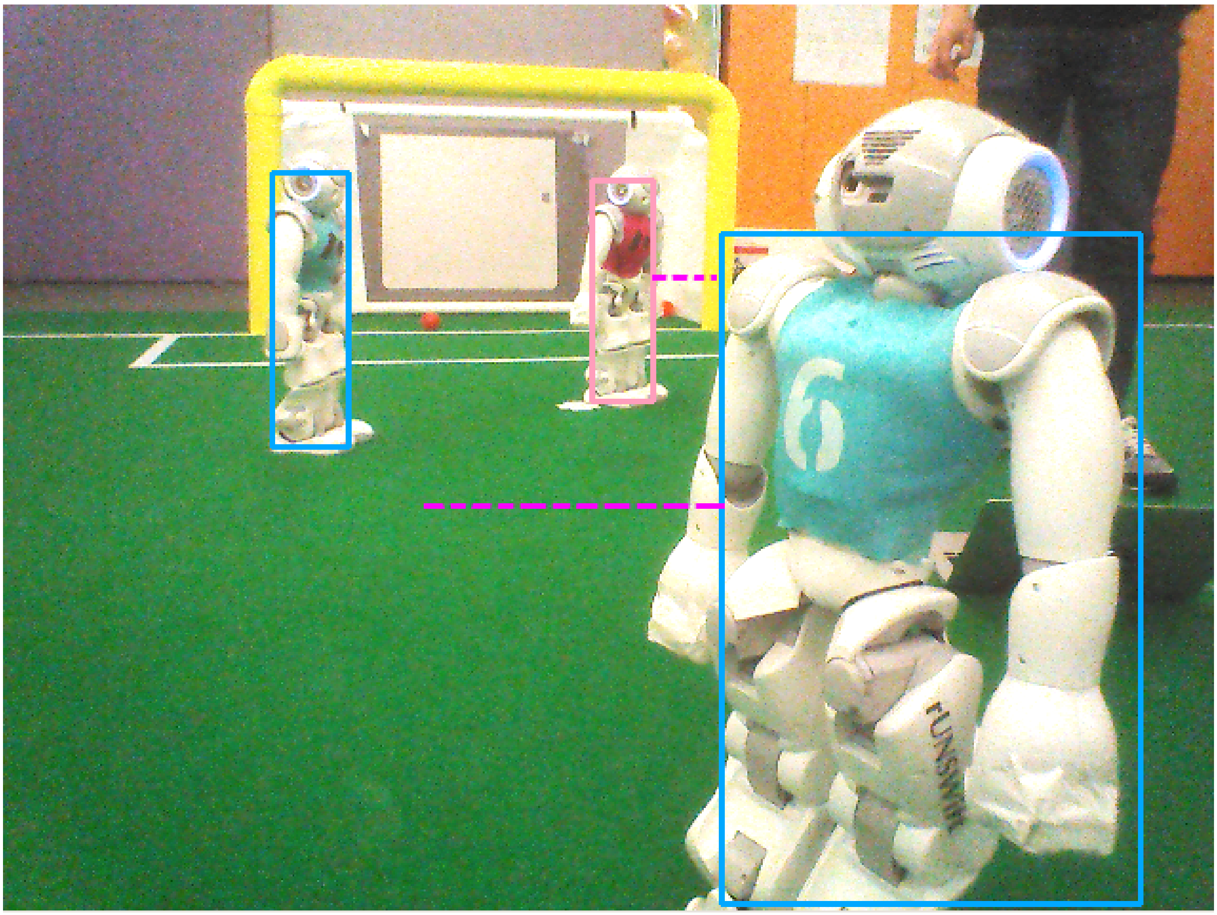

Figure 3.25: Half way line used for merging test

and hard to understand, the overall code is relatively simple. What it does is determine if this robot can be merged into a left robot or a right robot. If it can merge into both it will merge into the closest one. If it can only be merged into one, it will merge into that one and, if it cannot merge into any, it will move on.

Listing 3.3: Simplified code to determine when robots should be merged

```
std::vector<PossibleRobot> mergedRobots;
PossibleRobot merged;

for robot in possibleRobots {
    bool canMergeLeft = false;
    if (notFirst) {
        canMergeLeft = canMergePossibleRobots(merged, robot);
    }

    bool canMergeRight = false;
    if (notLast) {
        canMergeRight = canMergePossibleRobots(frame, saliency,
           robot, nextRobot);
    }

    if (canMergeLeft && canMergeRight) {
        int distanceLeft = boxDistance(merged.region, robot.
           region);
        int distanceRight = boxDistance(robot.region, nextRobot.
           region);

        //If right is closer than left
        if (distanceRight < distanceLeft) {
            mergedRobots.push_back(merged);
            merged = robot;

        //Left is closer
        } else {
            merged = mergePossibleRobots(merged, robot);
        }

    } else if (canMergeLeft) {
        merged = mergePossibleRobots(merged, robot);

    } else if (canMergeRight) {
        if (notFirst) {
            mergedRobots.push_back(merged);
        }
        merged = mergePossibleRobots(robot, possibleRobots[i +
                1]);

            //skip the next one
            i = i + 1;

        } else {
            if (notFirst) {
                mergedRobots.push_back(merged);
            }
            merged = robot;
        }
}

return mergedRobots;
```

Result

With this sample frame, the robot merging worked well as seen in Figure 3.26. It was able to merge the two legs together in the blue robot example and was able to join the arms to the body in the red robot example.

From running the merging code on a sample set of data, the following results were obtained:

- **Merged correctly**: 30
- **Merged too much**: 1
- **Merged too little**: 8
- $\therefore$ merge success = 76.9%

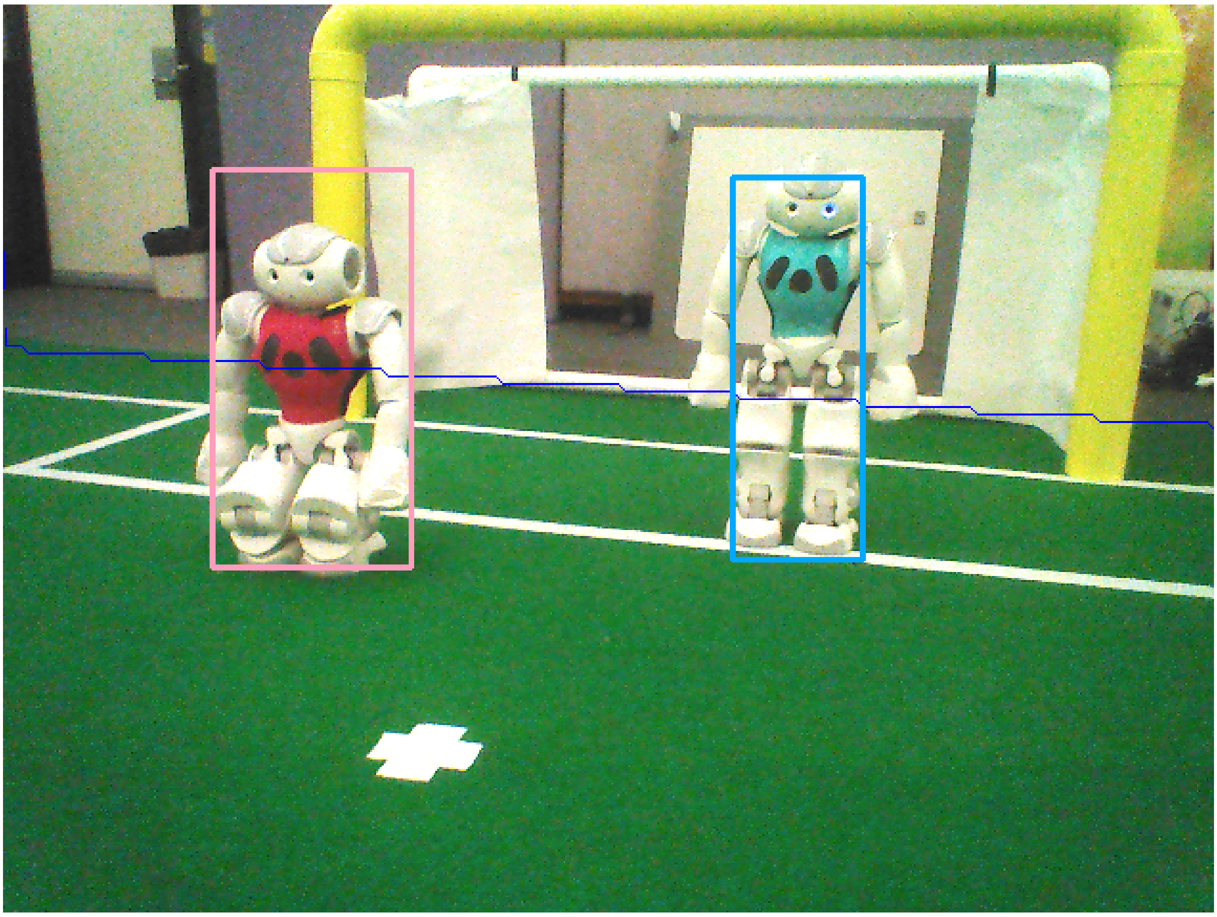

Figure 3.26: Result for robot merging

Analysis

From the results, it can be seen that the success rate for merging is 76.9%, which is not ideal. From the testing in this sample set, it can be seen that the algorithm can be more aggressive in merging as it rarely over-merges. Over-merging, however, can cause problems in terms of calculating the heading of an obstacle. This is seen in Figure 3.27 where the red robot is larger then it should be. The heading of the robot is calculated from the center of the box so overmerging results in the heading being incorrect.

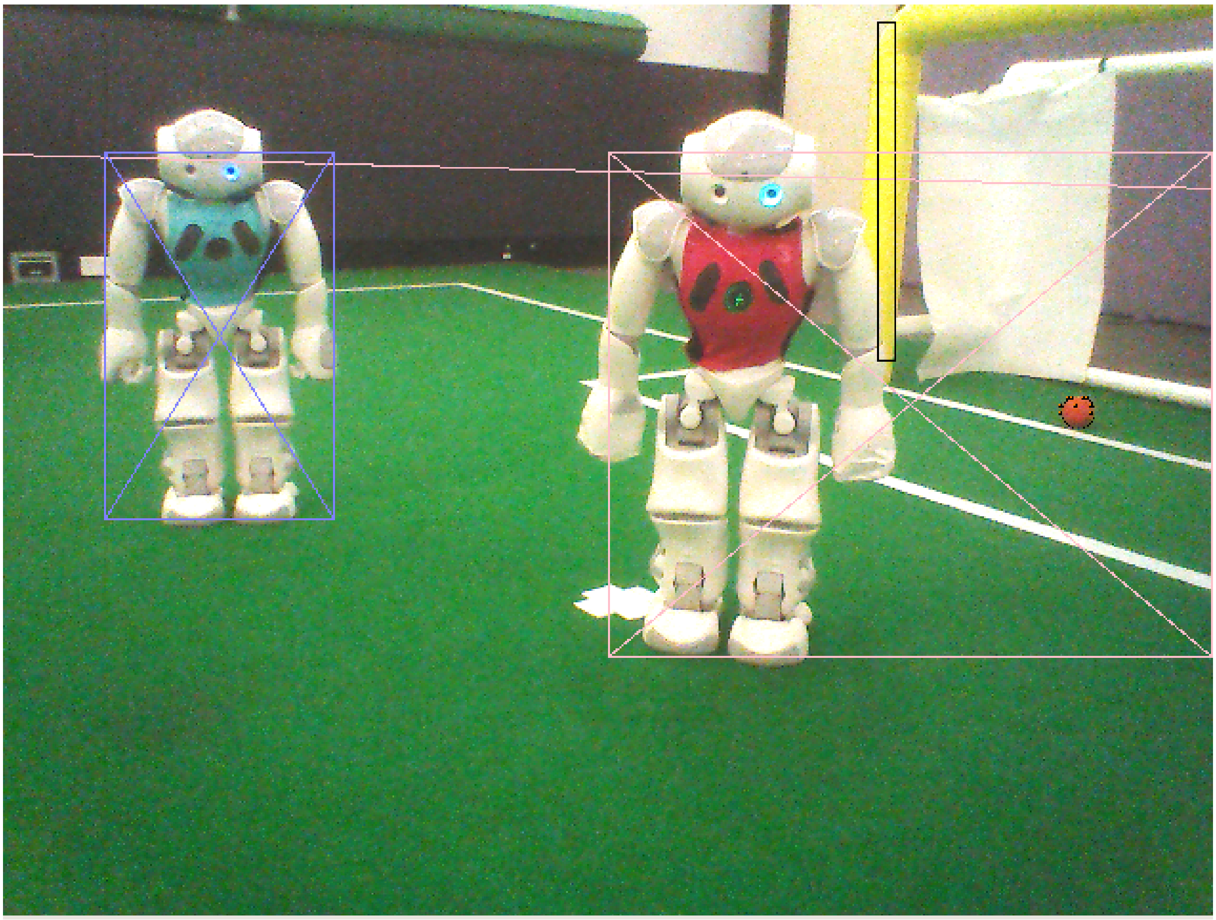

Figure 3.27: Over merging possible robots

3.2.8 Sanity Check Robots

Goal

The aim is to remove any false positives that may have slipped through. Examples include the determination that lines are robots or far-away objects are robots when they are not.

Implementation

These checks are fairly simple and only used to remove edge cases that may have slipped past. The first check done are to remove any robots detected over 4m away. At this distance, the robot detection sees more false positives then true positives so removing all of them is better than including them. This information, at the moment, is not useful for the overall game so is not deemed necessary to keep. If, in the future, a world model is implemented for robots ,then these far away objects could be useful and this sanity check could be removed.

The next sanity check involves removing oddly sized robots. For example, this removes robots that are very flat or very tall. It does this by calculating the gradient of the robot (rise/run). If the robot is not within a certain range, it is deemed an incorrect size. If the possible robot has a team, then the correct gradient range is increased to allow more through.

The last check is a more complicated implementation. The current code was experiencing problems when the goalie was seeing many false positives in the nets of the goals. This check is an attempt to help reduce the problem by taking the bottom of the obstacle and, if it is above all of the visible goal posts, it is considered a false positive and removed. The implementation of this check can be seen in Listing 3.4.

Listing 3.4: Simplified code to test whether a robot is above both posts

```
if (numberVisiblePosts == 1) {
    if (robot.y < post1.y) {
        return false
    } else {
        return true
    }
}

//Calculates the linear equation that joins the two posts and
   checks
//if the bottom of the robot is above this line.
else if (numberVisiblePosts == 2) {
    int postHeightDiff = abs(post1.y - post2.y);
    int postWidthDiff = abs(post1.x - post2.x);

    double m = (double)postHeightDiff / (double) postWidthDiff;

    int b = post1.y - m * post1.x;

    int postLineY = m * robot.x + b;
    if (robot.y < postLineY) {
        return false;
    } else {
        return true;
    }

} else {
    //Assumes numberVisiblePosts <= 2
    return true;
}
```

Results

When being run on sample set data, the sanity checker is able to remove false positives in the robot detection. For example, in Figure 3.28, we see two false positives being removed from the detection via the gradients of the detected robots. Figure 3.29 also shows some false positives being removed as they are above the goal posts and therefore should not be considered.

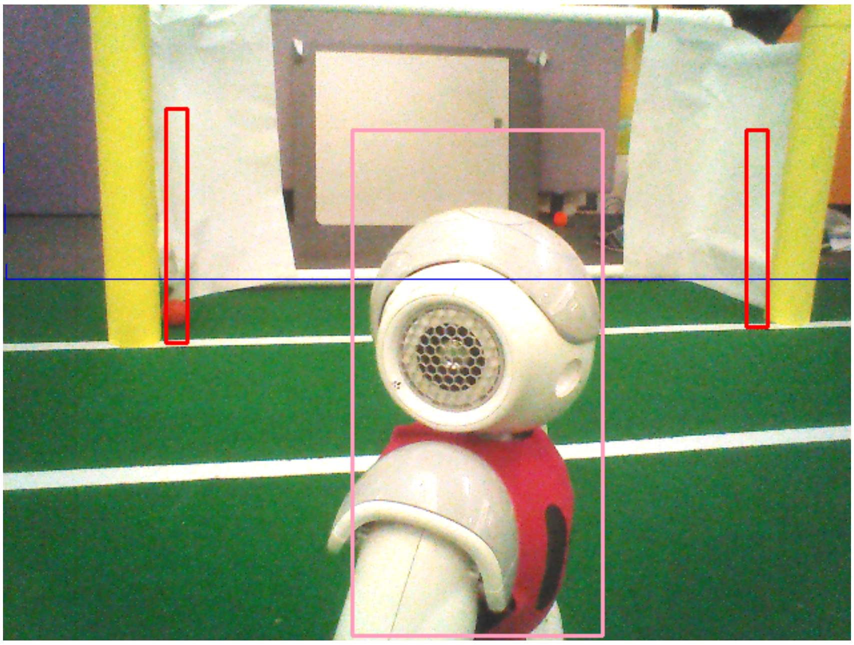

Figure 3.28: Sanity checker removing false positives via gradient

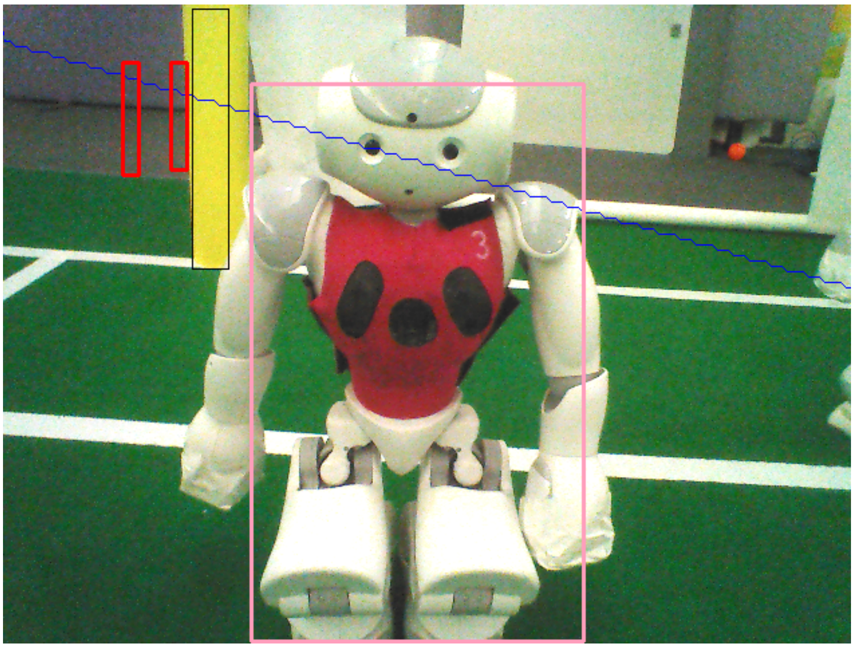

Figure 3.29: Sanity checker removing false positives above the goal post

3.2.9 Utilising the bottom camera

Goal

As well as using the top camera to detect robots, the information provided by the bottom camera is crucial for detecting close robots. The goal of this section is therefore to use the bottom camera with the top camera to detect obstacles.

Implementation

As the robot detection needs to be able to adequately detect close robots, a high significance needs to be placed on the bottom camera. A false negative is extremely detrimental to the robot as this could result in pushing the opponent robot. Therefore the implementation of the bottom camera detection utilises a modified code set to the top camera.

The detection steps are the same for the bottom camera as it applies the code for obstruction detection, feet detection, etc. However, one difference is that it does not apply the same sanity checks. For example, the Bayesian machine learning algorithm is not used, the team detection is not used and the sanity checks are not used. The bottom camera has its own distinct sanity checks that are more appropriate to that camera. For example, the only check that the bottom camera sanity checker makes is to ensure that it has the right colours, e.g. more than half robot colours.

From here, the top and bottom detections can be merged into a single obstacle. The way it does this is, after expanding the bottom and top possible robots, if they overlap, it tries to stitch these together. Therefore if they are around the same spot in the horizontal space they are assigned as the same robot and merged. This can be seen in Figure 3.30 where we can see the robots in the top and bottom camera being merged. The way you can clearly see this merging is that the bottom camera has assigned the feet as blue even though it cannot see the jersey.

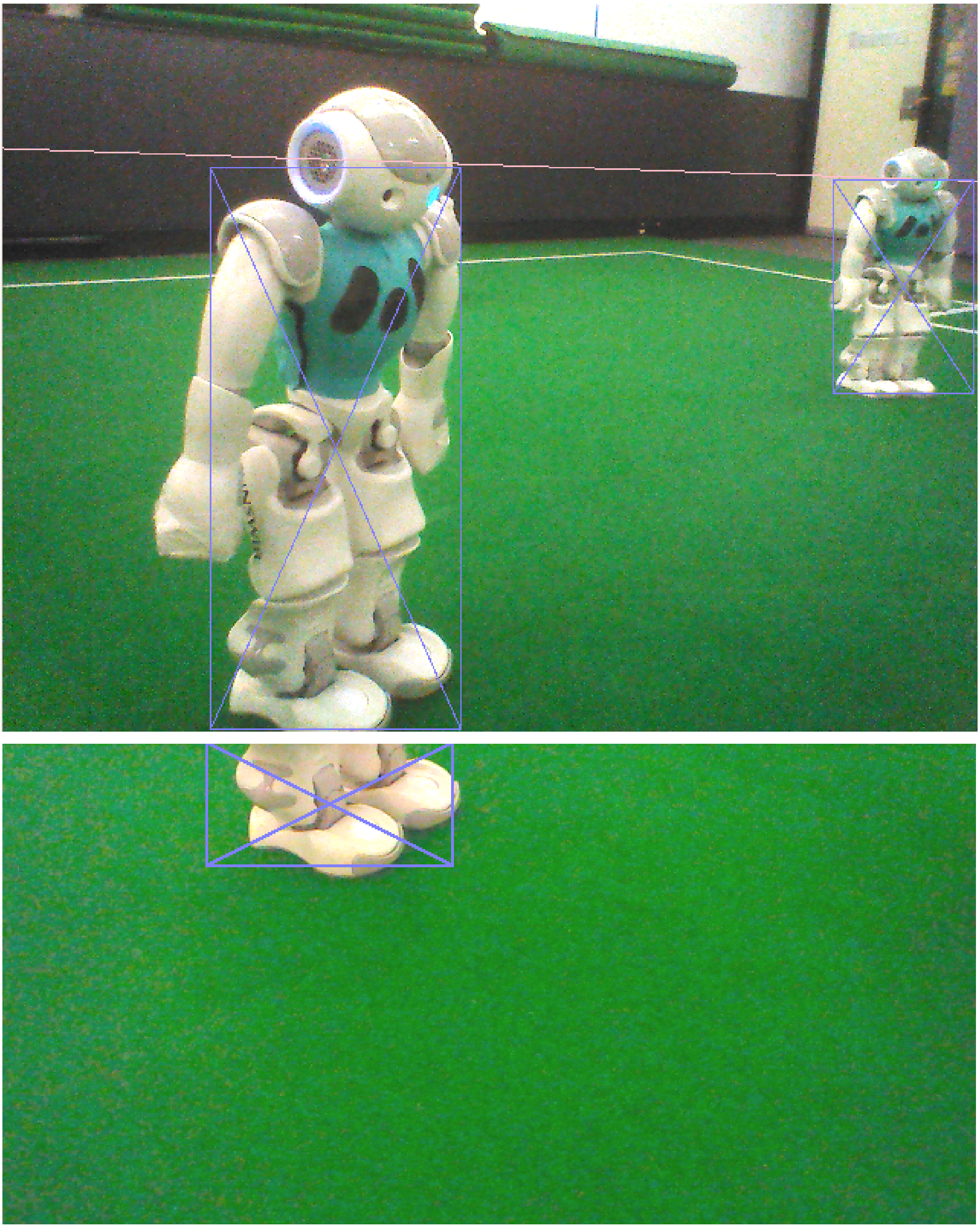

Figure 3.30: Top and bottom camera stitching possible robots

Results and analysis

From experimenting it was seen that the top and bottom camera could be utilised quite effectively to create a merged observation. The result of this was a better calculation of the distance of the robot as it is able to correctly determine the bottom of the feet and therefore calculate the distance with higher accuracy. It was also able to detect obstacles in the bottom camera even when the top camera struggled as seen in Figure 3.31.

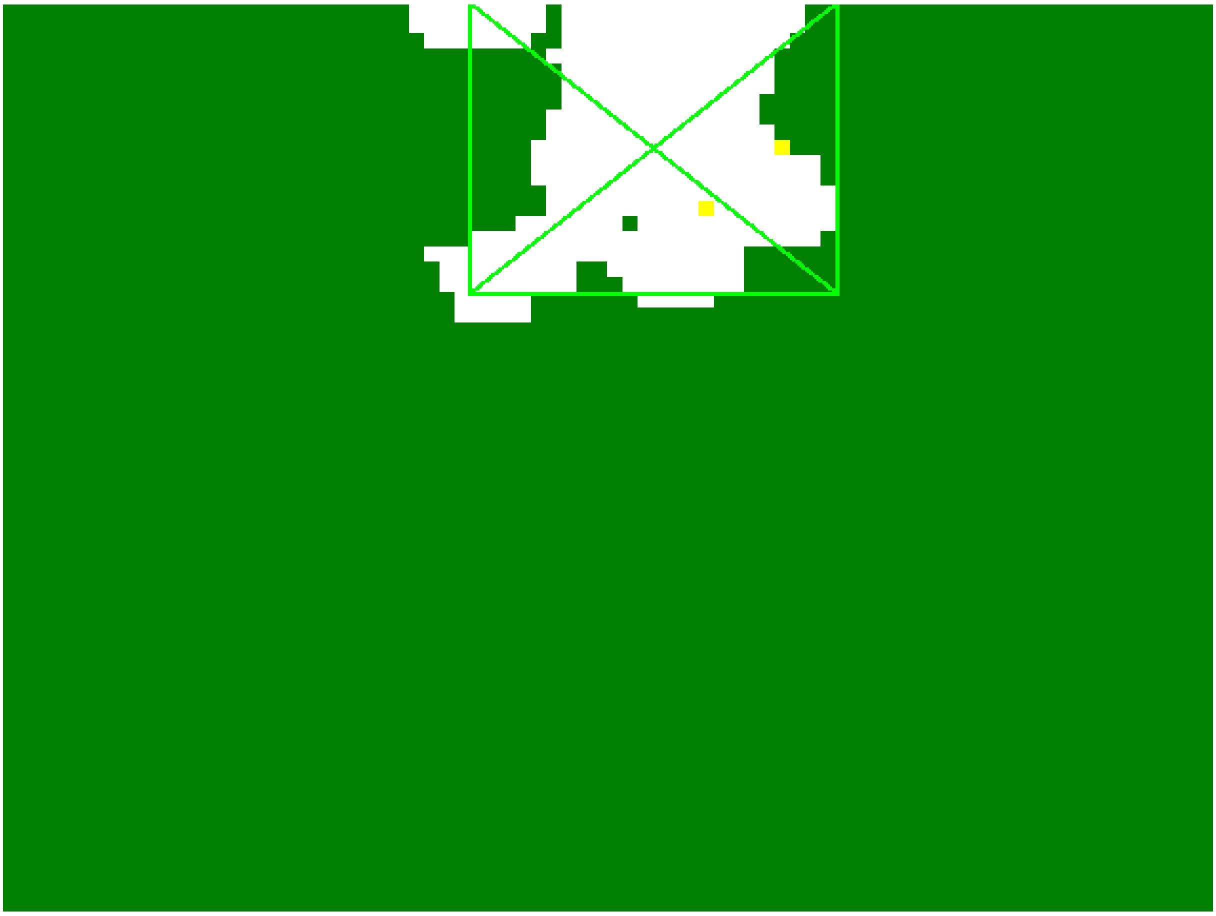

Figure 3.31: Bottom camera being used only to detect obstacles

3.3 Results

3.3.1 Sample data 1

This includes the results for the sample data that have been used to test each individual component of the robot detection.

Result	Count
Success	64
Success no team	0
Success wrong team	0
False positive	7
False negative	24
Bad box placement	5

$$\begin{aligned}\text{Success detecting obstacle} &= \frac{64}{64+7+24+5} \\ &= 64\%\end{aligned} \tag{3.11}$$

Table 3.6: Old robot detection through a sample data set

Result	Count
Success	80
Success no team	5
Success wrong team	2
False positive	9
False negative	3
Bad box placement	1

$$\begin{aligned}\text{Success detecting obstacle} &= \frac{80+5+2}{80+5+2+9+3+1} \\ &= 87\%\end{aligned} \tag{3.12}$$

Table 3.7: New robot detection through a sample data set

3.3.2 Moving data set

This data set is to represent what the robot detection is like during a game in which the robot is moving a lot.

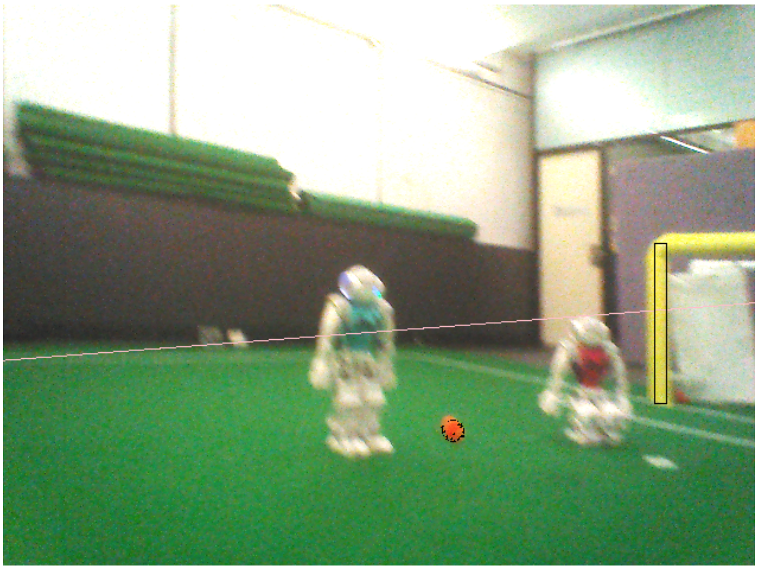

Figure 3.32: Old robot detection for moving frame 1

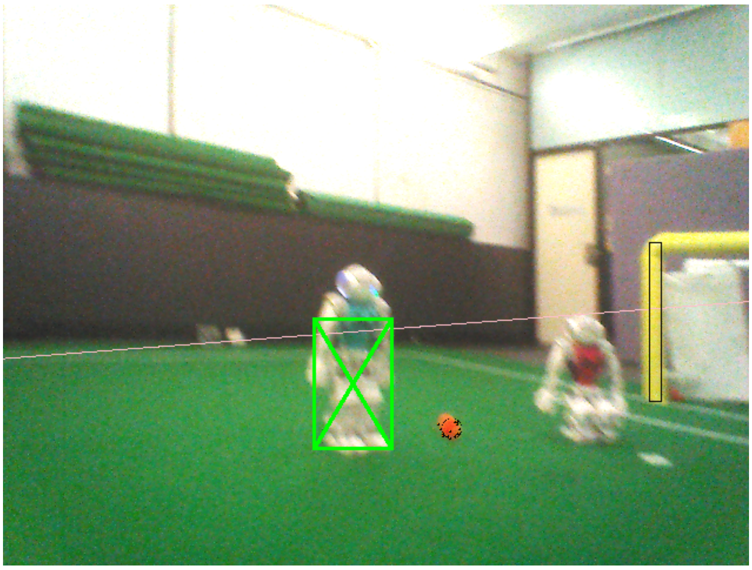

Figure 3.33: New robot detection for moving frame 1

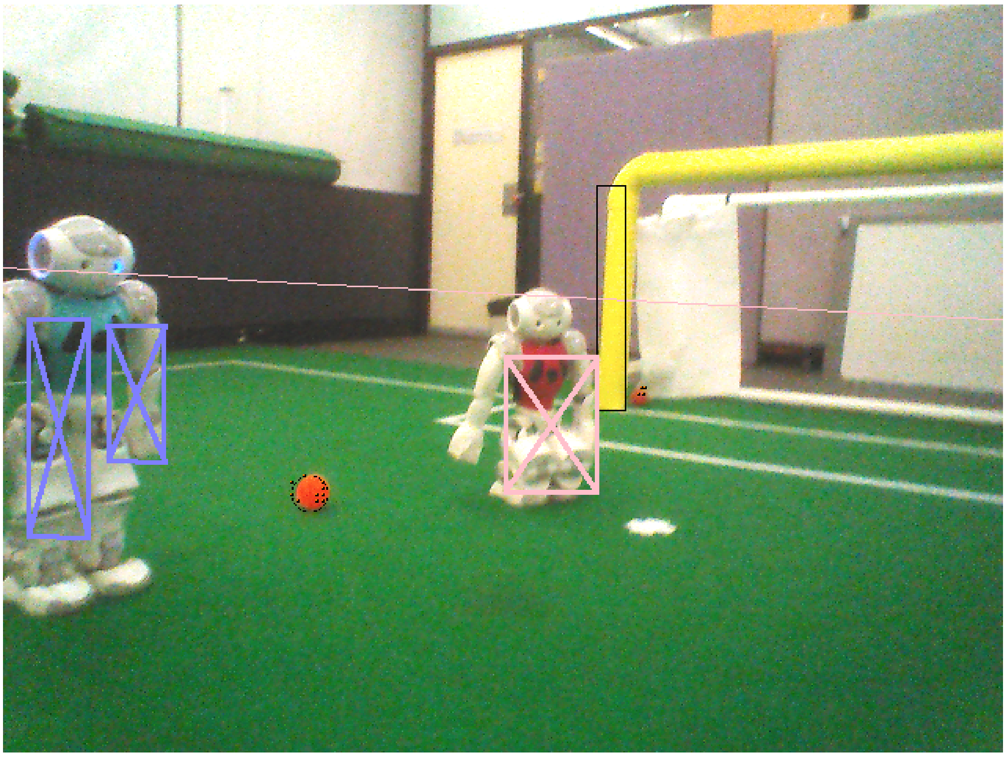

Figure 3.34: Old robot detection for moving frame 2

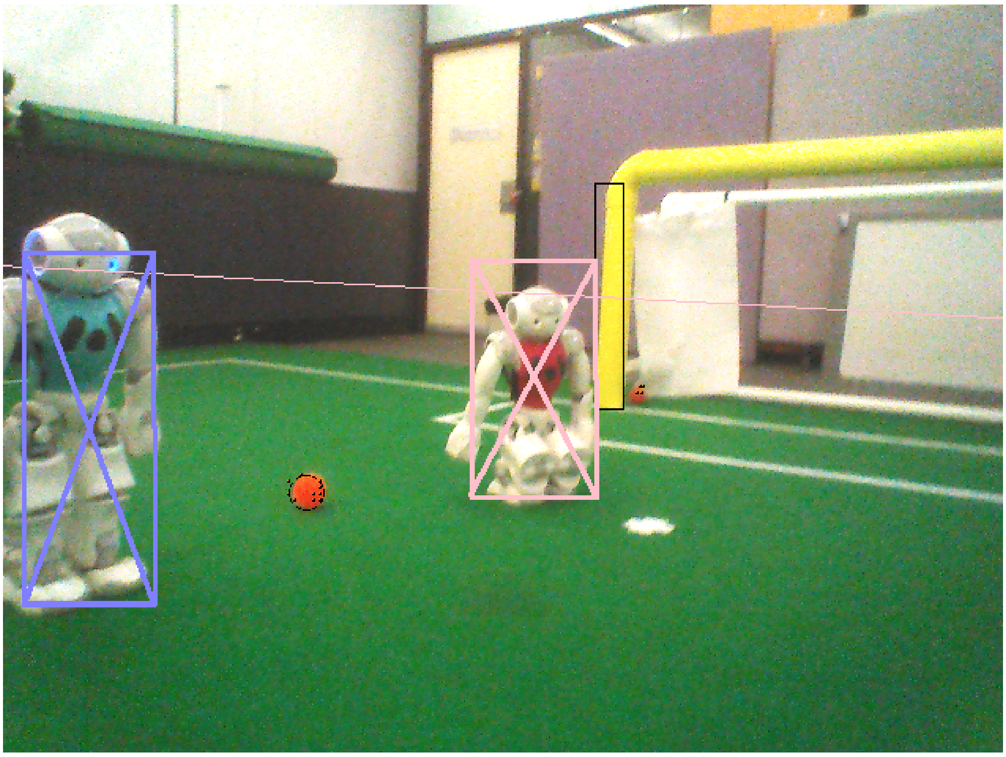

Figure 3.35: New robot detection for moving frame 2

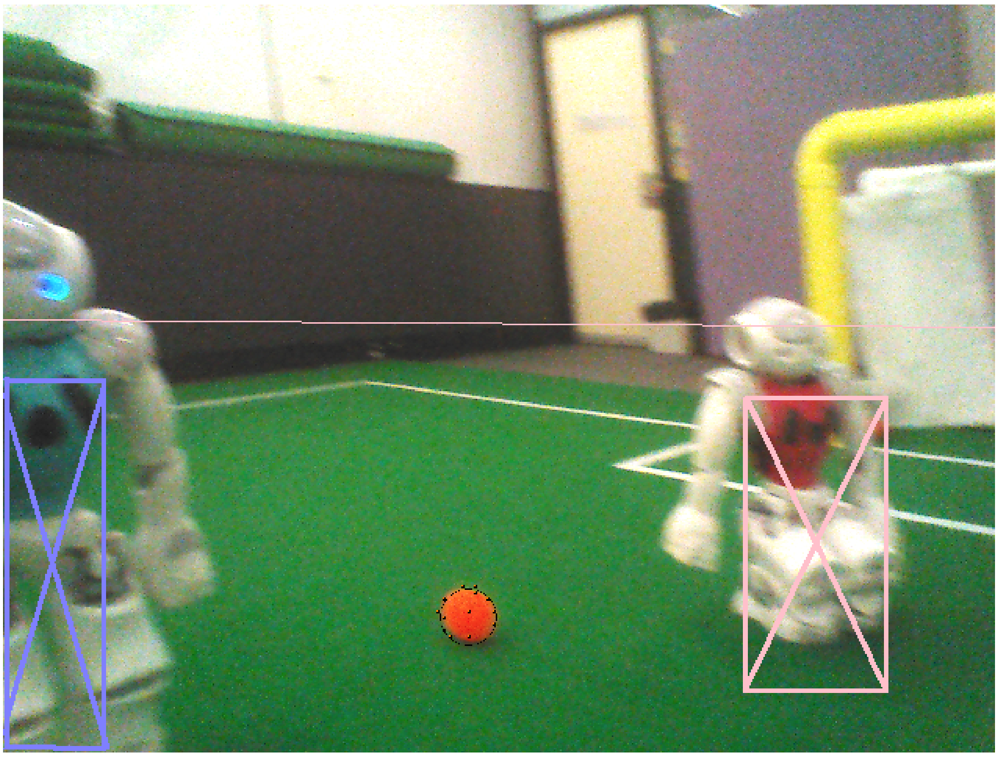

Figure 3.36: Old robot detection for moving frame 3

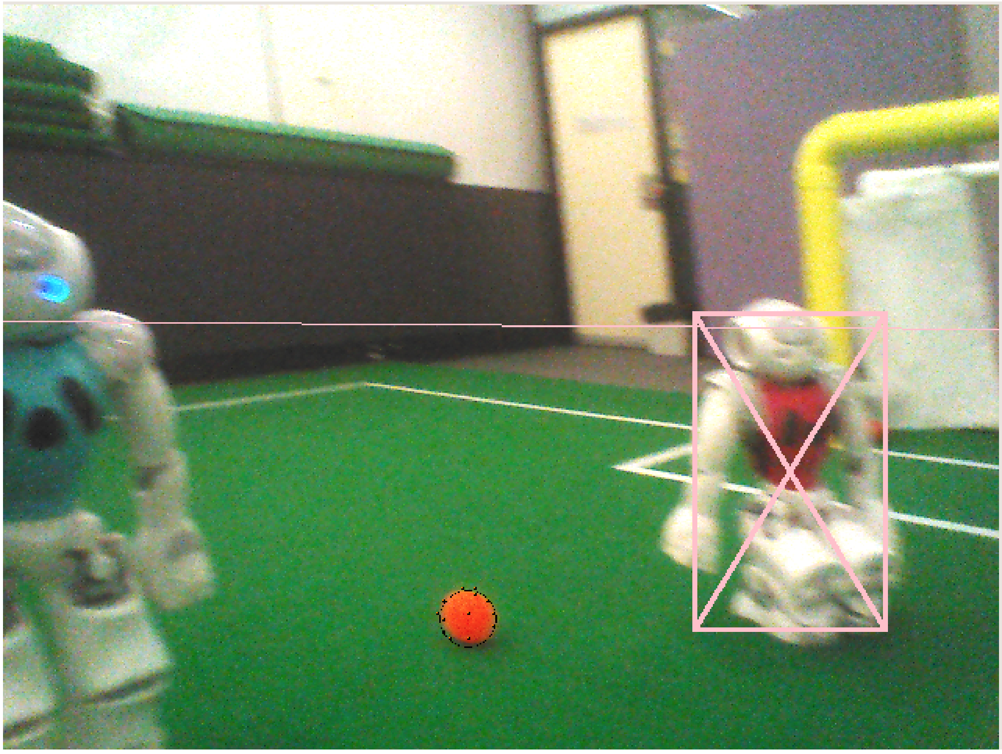

Figure 3.37: New robot detection for moving frame 3

3.3.3 Larger sample set

Result	Count
Success	85
Success no team	0
Success wrong team	15
False positive	8
False negative	57
Bad box placement	2

$$\begin{aligned}\text{Success detecting obstacle} &= \frac{85+0+15}{80+0+15+8+57+2}\\ &= \frac{100}{167}\\ &= 59.9\%\end{aligned} \tag{3.13}$$

Table 3.8: Old robot detection through a larger sample data set

Result	Count
Success	116
Success no team	17
Success wrong team	0
False positive	2
False negative	27
Bad box placement	3

$$\begin{aligned}\text{Success detecting obstacle} &= \frac{116+17+0}{116+17+0+2+27+3}\\ &= \frac{133}{165}\\ &= 80.6\%\end{aligned} \tag{3.14}$$

Table 3.9: New robot detection through a larger sample data set

3.3.4 Visual results

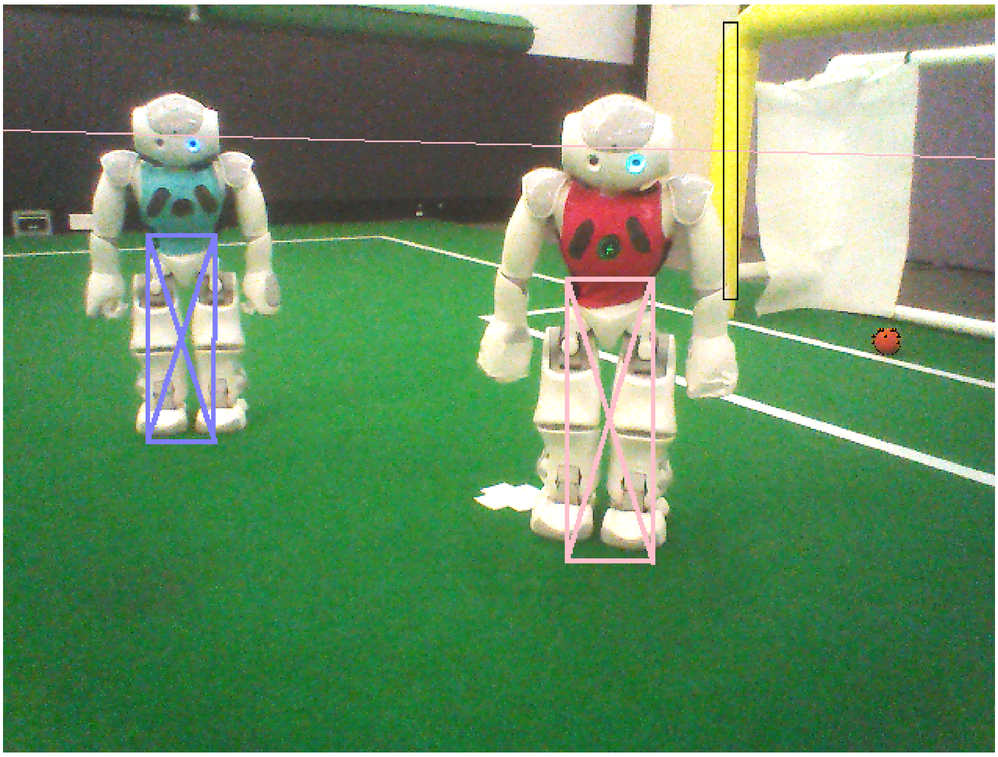

Figure 3.38: Old robot detection results 1

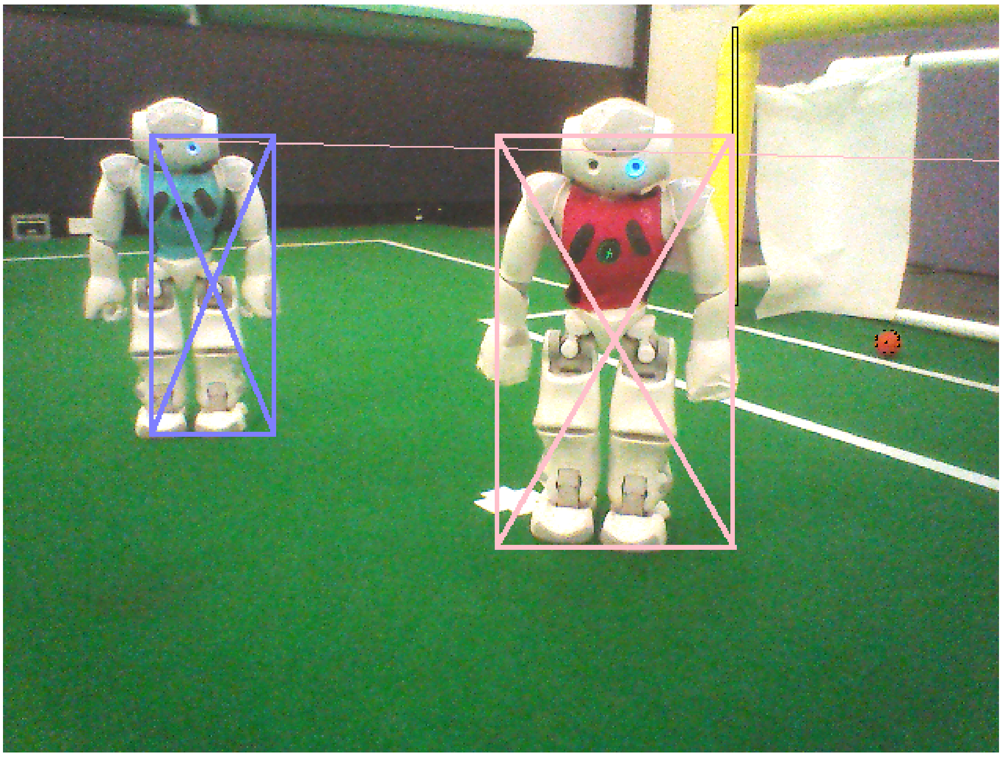

Figure 3.39: New robot detection results 1

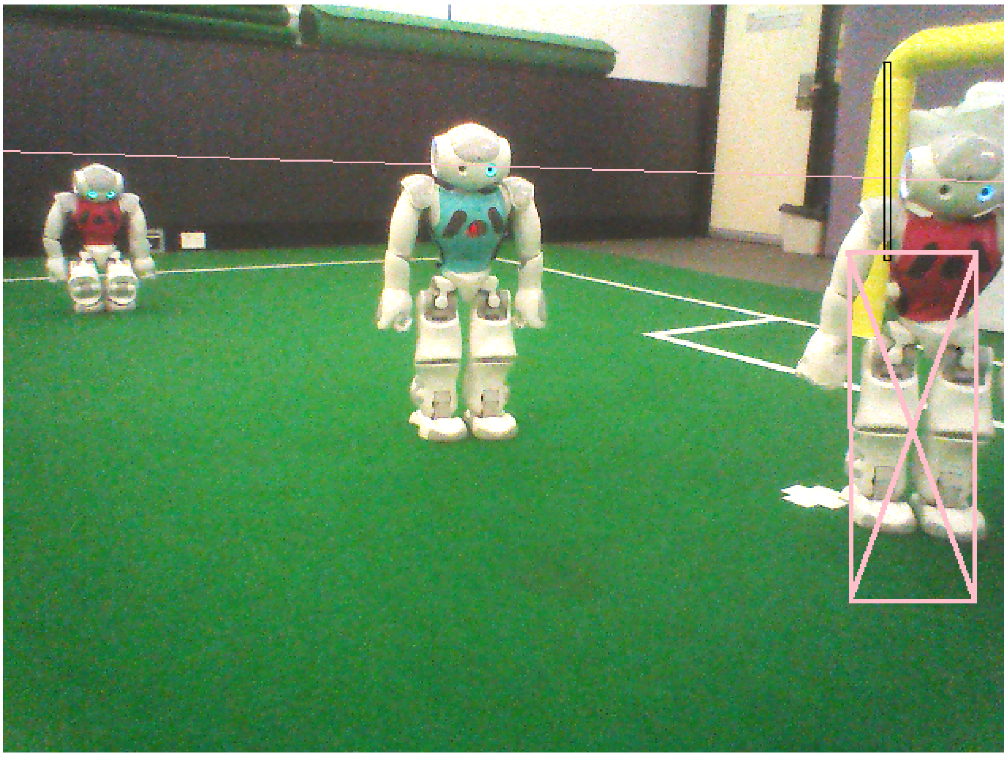

Figure 3.40: Old robot detection results 2

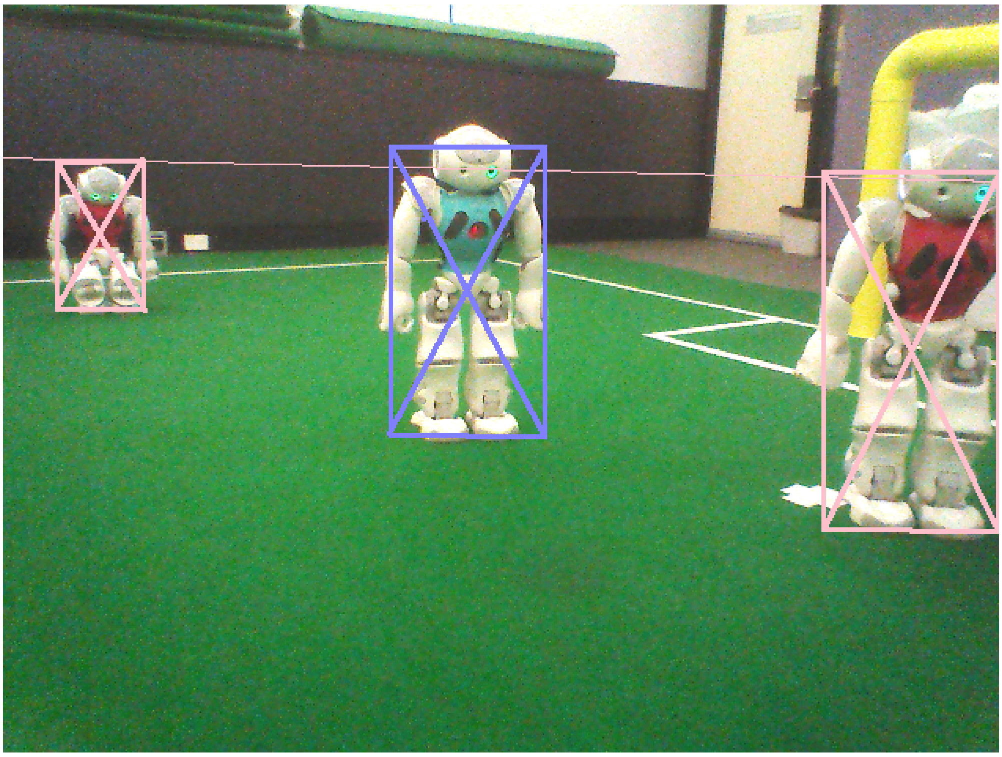

Figure 3.41: New robot detection results 2

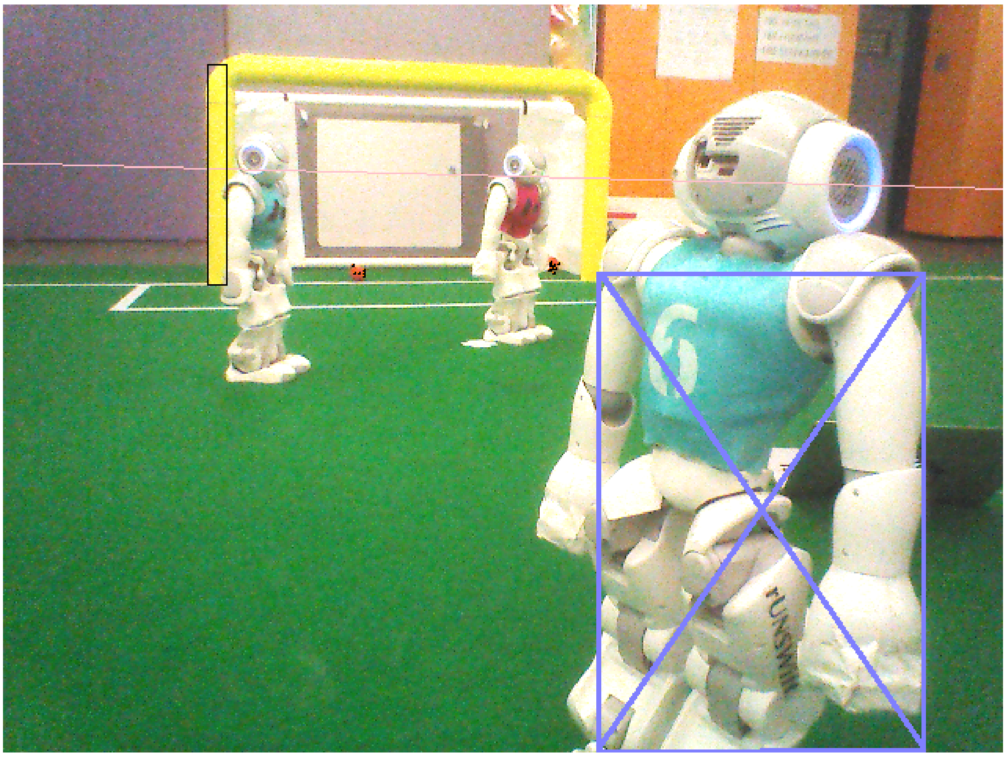

Figure 3.42: Old robot detection results 3

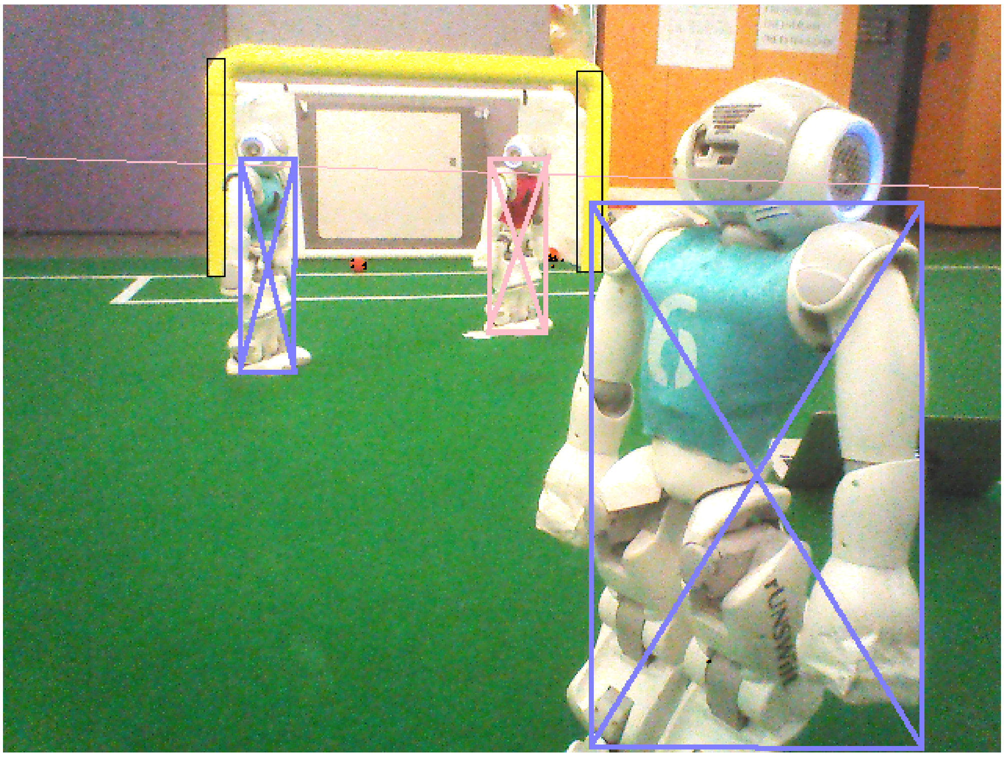

Figure 3.43: New robot detection results 3

3.4 Evaluation

From the results seen in Section 3.3, the new robot detection does significantly better than the old. In the sample set used in each of the sub-component's results, the old robot detection achieves success rate of 64% while the new achieves 87%. It can be seen that the old robot detection is not very aggressive in assigning a robot with 24 false negatives to the new system's 3; however, it has fewer false positives with 7 compared to the new system's 9. As this is not significantly fewer, it can be deduced that the increased desire to positively detect a robot is beneficial.

Both algorithms were passed into a moving sample set, which is meant to represent the sample input that is experienced through a game. The motion causes motion blur which can negatively impact the visual algorithms. This is most notably a problem when it is doing large turns of the head or body. Some of the frames in this sample set can be seen in Section 3.3.2. Examining these frames, it appears as if the new detection is doing marginally better then the old robot detection. It is able to distinguish the obstacles well when it is looking directly at them, while the old detection was having some trouble.

The algorithms were tested on another sample set and the results were similar with the new robot detection coming out on top. The new robot detection had a success rate of 80.6% compared to the old system's 59.9%. You can clearly see the improvements of the robot detection code in the visual frames shown in Section 3.3.4 where the new algorithm is correctly detecting all robots.

In conclusion, it can be seen that the new robot detection is a significant improvement over the old.

Chapter 4

Robot Filter

4.1 Introduction

After the robots have been detected visually, information about these obstacles needs to be provided to the behaviour modules. Raw visual information is not enough due to the robot's fluctuating distances and heading so the robot filter is a method to group consecutive readings together into a single obstacle.

4.2 Goal

As the robot observations calculation has a margin of error, and as the robots are moving on the field, it is not easy to join what is currently being seen with what was seen in the past. The goal of the robot filter is to take any new observations and try and merge these with previous frames. Therefore, if the robot sees a robot moving across the screen, the behaviour module should get information about a single robot moving at the same rate as it is in real life. This should result in generating an accurate model of what the field is like in regards to where robots are positioned.

4.3 Current implementation

The current robot filter was implemented in previous years and had been used by the behaviour module. When tested against the new robot detection code it was seen that it did poorly in grouping the observations together. The result was a huge cluster of robot obstacles when in fact there should only be a few. This can be seen in Figure 4.1.

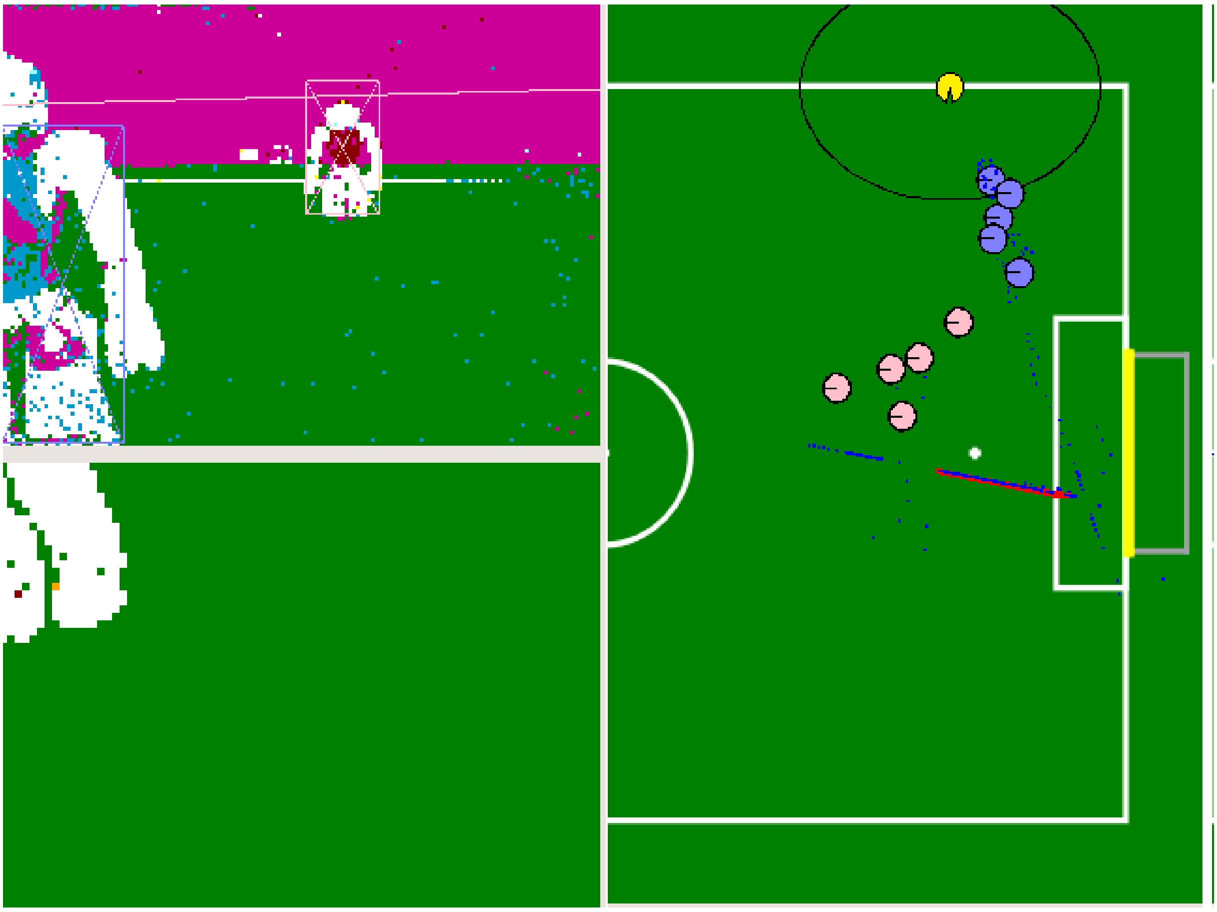

Figure 4.1: Old robot filter failing to merge observations while moving

The preliminary guess as to why this problem was abundant with the new robot detection was because of its improved ability to detect far away obstacles. Observations further away are more prone to margins of error in distance or heading, which the current robot filter does not seem to be able to handle. Another possible explanation is that the robot filter never worked that well. It is not clear how reliant the current behaviour code was on these obstacles so in fact it could have been doing poorly all along.

In view of these failures, it could be desirable to fix the robot filter to better cluster consecutive robot observations into a single obstacle. This is important as the behaviour code is extremely dependent on having a good understanding of where obstacles are on the field.

4.4 Implementation

The new implementation of the robot filter divided the logic into three separate modules. These included:

- **Robot Observation**: these are the raw visual observations. This module will store the visual information and contain logic on whether the observation is still valid.
- **Robot Group**: these are the clusters of robot observations over time and include logic such as whether it is possible to merge an observation into the group.
- **Robot Filter**: this includes the logic of determining when an observation should be merged into a group. This calculates what groups the observation can be merged into and decides which one it will do.

The reason for splitting it up into separate modules was to increase code readability and to make testing easier.

4.4.1 Robot Observation

When a robot observation is created, it is assigned a life score, which is an arbitrary value, say 120, used to determine whether the observation should still be considered valid. When the observation's life score reaches zero it is determined to be stale and can be removed. This is done so that observations are eventually considered too old to be useful.

There are two main metrics used to reduce the lifescore:

1. **Time**: as time goes by the lifescore should reduce.
2. **Vision**: if the group is in sight it should be reduced quicker if it is not in sight.

When the lifescore of an observation is being reduced over time, its rate should be scaled if the obstacle is in view. For example, if the robot sees an object directly in front of it but fails to see it again when continuing to look at the same spot, the lifescore reduction should be large so it gets destroyed quickly. However, if an obstacle is continues to be seen, then the obstacle's lifescore should still be reduced quickly but the new observations will keep it alive.

The other consideration when reducing the life score over time is a robot obstacle that is now off-screen. Obstacles detected previously that are now off-screen should remain in the list of detected obstacles for a longer time so that there is information about previously detected robots. The way that this is done is by not reducing the lifescore as heavily if the robot group is no longer on-screen. If a robot is visually detected and the robot turns away from it, the robot filter should still remember it for a certain period. If it turns back to where that obstacle used to be and no longer sees it, the life score will be heavily reduced due to it not being on screen, as discussed above.

Some magic numbers were used to determine how long an observation should stay valid if it is on-screen and off-screen. In the current implementation observations have been set to expire after 1s if it is on-screen and 10s if it is off-screen. These can easily be changed by modifying the magic number definitions in the program.

One assumption made is that the camera is running at a steady 20FPS (obtained by looking at the average framerate while running). This is used to calculate the max lifescore by multiplying the framerate by the number of seconds that the observation should remain valid off-screen. The next phase is defining the rate that the lifescore will be reduced if it is onscreen or off-screen. The off-screen reducer is set at 1 as this will result in the lifescore reaching zero exactly as the seconds for off-screen defined. The on-screen reducer is set at how much longer the observation would remain alive off-screen compared to on-screen. For example, if on-screen was 2s and off-screen was 8s, the off-screen reducer would be 1 and the on-screen reducer would be 4 and the max lifescore would be 160. However, in the current implementation, the off-screen reducer is 1, on-screen reducer 10 and max lifescore 200. The implementation can be seen in Listing 4.1.

Listing 4.1: Code showing how to calculate lifescore variable values

```
AVERAGE_FRAME_RATE = 20
SECONDS_ALIVE_ONSCREEN = 1
SECONDS_ALIVE_off−screen = 10

MAX_LIFE_SCORE = SECONDS_ALIVE_OFFSCREEN * AVERAGE_FRAME_RATE
SECONDS_ALIVE_RATIO = SECONDS_ALIVE_OFFSCREEN / SECONDS_ALIVE_ONSCREEN

 off−screen_REDUCER = 1
ONSCREEN_REDUCER = SECONDS_ALIVE_RATIO
```

4.4.2 Grouped Robots

The next level of the robot filter represents the group or cluster of observations. It groups together multiple observations and creates a coordinate which is suppose to represent some sort of grouped average. This also includes logic on whether an observation can be merged into it.

Merging logic

The simplest logic that could be used is creating a circle around the group allowing an observation to be merged if it is within this circle. This can be done quite simply using Cartesian coordinates for the group and observation. Figure 4.2 shows an observation which should be merged into the group as it is less than the merge radius.

The group calculates whether an observation is within the radius by checking that the distance from the group to the observation is less then the radius. To calculate the distance the group and observation are represented as Cartesian coordinates as seen in Figure 4.3 and Pythagoras is used to calculate the hypotenuse(distance) as seen in Equation 4.1. In this example, the coordinates (x_1, y_1) for the group and (x_2, y_2) for the observation are used.

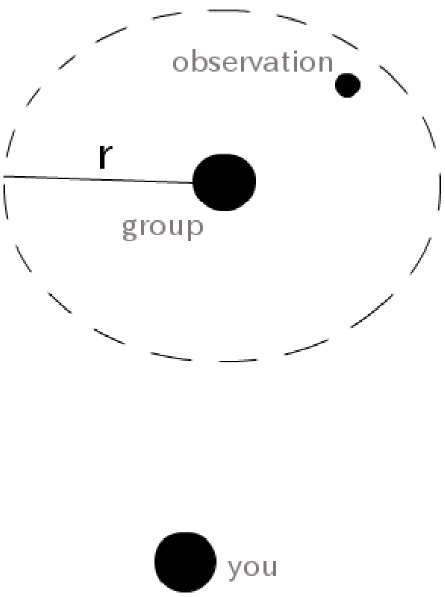

Figure 4.2: Robot filter observation to be merged into group

$$
\begin{aligned}
x_{diff} &= x_1 - x_2 \\
y_{diff} &= y_1 - y_2 \\
distance^2 &= x_{diff}^2 + y_{diff}^2 \\
distance &= +\sqrt{x_{diff}^2 + y_{diff}^2}
\end{aligned}
\tag{4.1}
$$

From experimentation, a circle for merging observations is not good enough in practise because of its inability to cater for robot observations with larger margins of error in distance compared to their heading. Therefore, when determining whether an observation can be merged, the algorithm needs to be more lenient on am observation's vertical distance compared to its horizontal. This can be seen in Figure 4.4 where the vertical radius is larger then the horizontal.

While there is a desire to improve the merging logic, the performance of the algorithm is also important as it should not increase the processing intensity significantly compared to the circular approach. The way this is done is by scaling the larger radius to generate a circle around the robot and doing the same calculation as before. This compression can be seen in Figure 4.5. This compression technique is similar to a transform tool in a photo editor, where one axis of the image is compressed. From compressing it by a

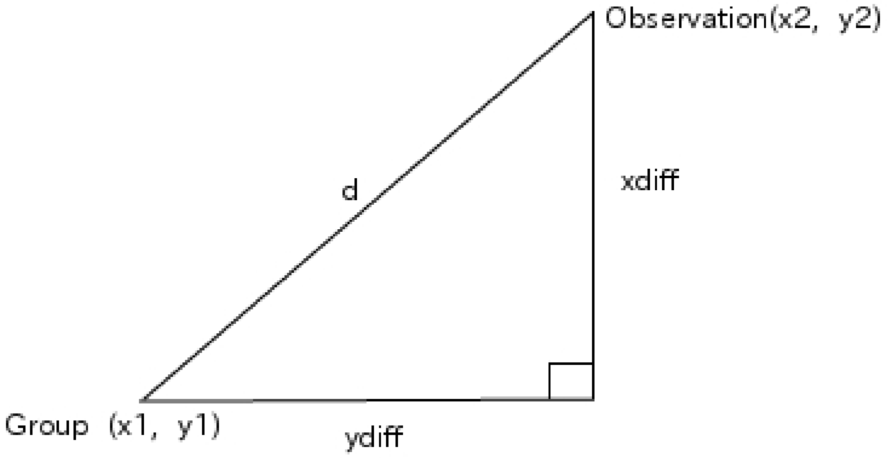

Figure 4.3: Triangle representing the distance from group to an observation

certain amount, a circle is able to be created and the original calculation can be done.

The maths for doing the scaling is seen in Equation 4.2 where only the vertical difference of the image is scaled.

$$
\begin{aligned}
scale\_ratio &= \frac{r_1}{r_2} \\
x_{diff} &= (x_1 - x_2) * scale\_ratio \\
y_{diff} &= y_1 - y_2 \\
distance^2 &= x_{diff}^2 + y_{diff}^2 \\
distance &= +\sqrt{x_{diff}^2 + y_{diff}^2}
\end{aligned}
\tag{4.2}
$$

Another component placed into this algorithm is a time-based scaling for the merging

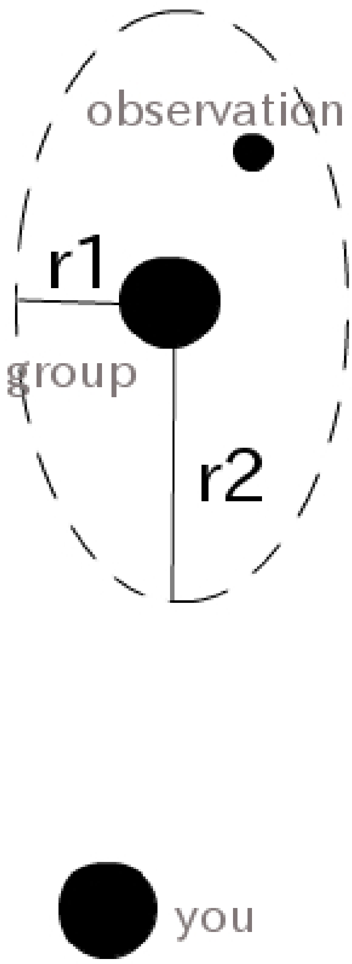

Figure 4.4: Merge radius via an eclipse not a circle

distance. If it has been a while since there has been a fresh observation for the group, the merging radius slowly scales larger. This is because the staler the observations, the more margin of error the coordinates have.

The algorithm uses a simple linear scale factor. If it just received a frame, the radius scaling is 1, therefore not changing, or if it is the longest period it is scaled by 2. This means that the radius of the ellipse can be twice as large as its original radius.

Coordinate generation logic

When there is a certain number of observations in the group, a method to generate a coordinate, which represents a good average of the group, is needed. A simple average

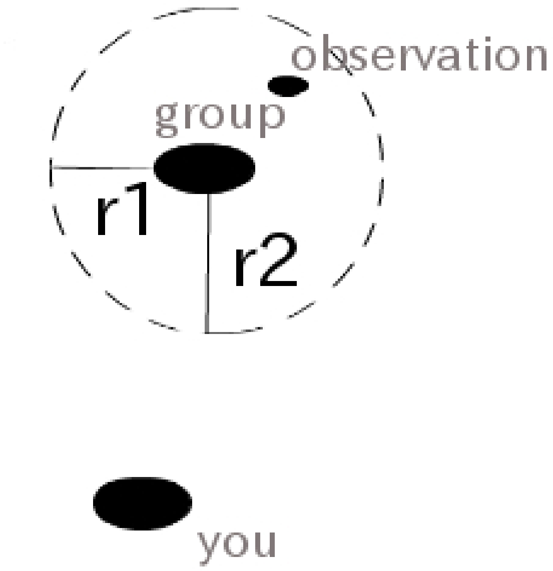

Figure 4.5: Merge radius via an eclipse compressed to create a circle

algorithm could be used, which averages each observation equally. The problem with this is that new observations should have more weight and are more likely to be correct then an observation seen 10 frames ago. The weight calculation system seen in Listing 4.2 completes this calculation.

Listing 4.2: Calculating the coordinates via weights

```
groupX = 0;
groupY = 0;
totalWeight = 0;

foreach (observation in observations)
```

```
        weight = observation.getWeight();

        coordinates = observation.getCartesianCoordinates();
        groupX += (weight * coordinates[0]);
        groupY += (weight * coordinates[1]);

        totalWeight += weight;
    }

    if (totalWeight != 0) {
        groupX /= totalWeight;
        groupY /= totalWeight;
    }
```

4.4.3 Robot Filter

The robot filter is used to determine which observations are merged into which group. An observation might be able to be merged into multiple groups so the robot filter's goal is to distribute the observations optimally. One condition is that multiple observations cannot go into a single group.

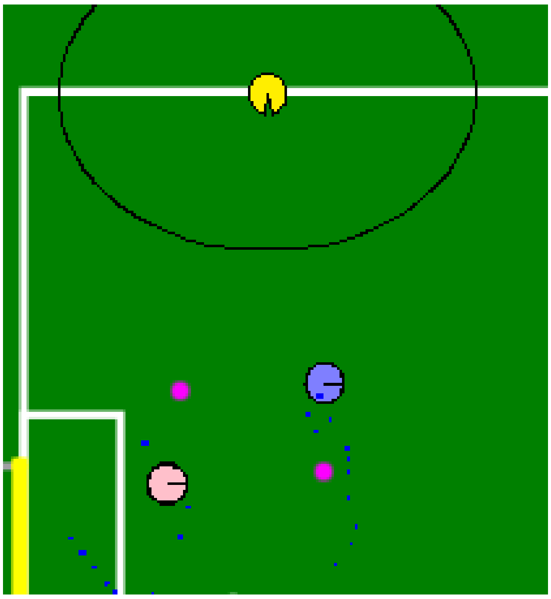

Figure 4.6: Example where we have two observations(purple) to be merged into two groups

For example, consider a situation similar to the one shown in Figure 4.6 where there are two observations, indicated by purple, that are to be merged into the two groups. One possible solution is to find the length of each observation to each group and choose the combination that results in the minimum overall distance between observations and groups. This would result in an optimal solution and also it would not be too CPU intensive as it only does 4 calculations. However, as the number of observations and groups increases, the CPU intensity becomes extremely high. If there was a limit of 10 groups and 10 observations in one frame - unlikely but worst case - the number of calculations would be 10P10, which is 10! = 3,628,800 calculations for a single frame. A factorial calculation is not desired and so a different algorithm should be implemented.

The final algorithm that was implemented was a suboptimal solution that does well enough. For each group, it looks at every observation that has not currently been merged into a group. Of all the observations, it picks the one with the shortest distance that is able to be merged into the group. It merges that observation into the group and marks it as merged. Any observations that were not merged into a group create their own groups. This can result in an observation that would have been better to be merged into a different group but it is a good enough solution that is not CPU intensive. For example, if there are K groups and J observations the Big O complexity would be J * K.

Listing 4.3: Merging observations into groups

```
foreach(group in groupedRobots) {
    int smallestIndex = NO_CLOSE_OBSERVATION;
    double smallestDistance = 0;
    foreach(observationIndex in observations) {

        if (!observationMerged[observationIndex]) {
            observation = observations[observationIndex]

            if (group.canMergeRobot(observation)) {

                double distance = group.distanceToRobot(
                    observation);
                if (smallestIndex == NO_CLOSE_OBSERVATION ||
                     smallestDistance > distance) {
```

```
                    smallestIndex = observationIndex;
                    smallestDistance = distance;
                }
            }
        }
    }

    if (smallestIndex != NO_CLOSE_OBSERVATION) {
        observation = observations[smallestIndex];
        group.mergeRobot(observation);
        observationMerged[smallestIndex] = true;
    }
}

foreach(observationIndex in observations) {
    if (!observationMerged[observationIndex]) {
        groupedRobots.push_back(new Group(observations[
           observationIndex]));
    }
}
```

4.5 Results

4.5.1 Laboratory

From running the new robot filter in game scenarios and in standing positions, it was determined that it was performing well as seen in Figures 4.7 and 4.8.

This is compared to some of the results that were produced by the old robot filter as seen in Figure 4.1.

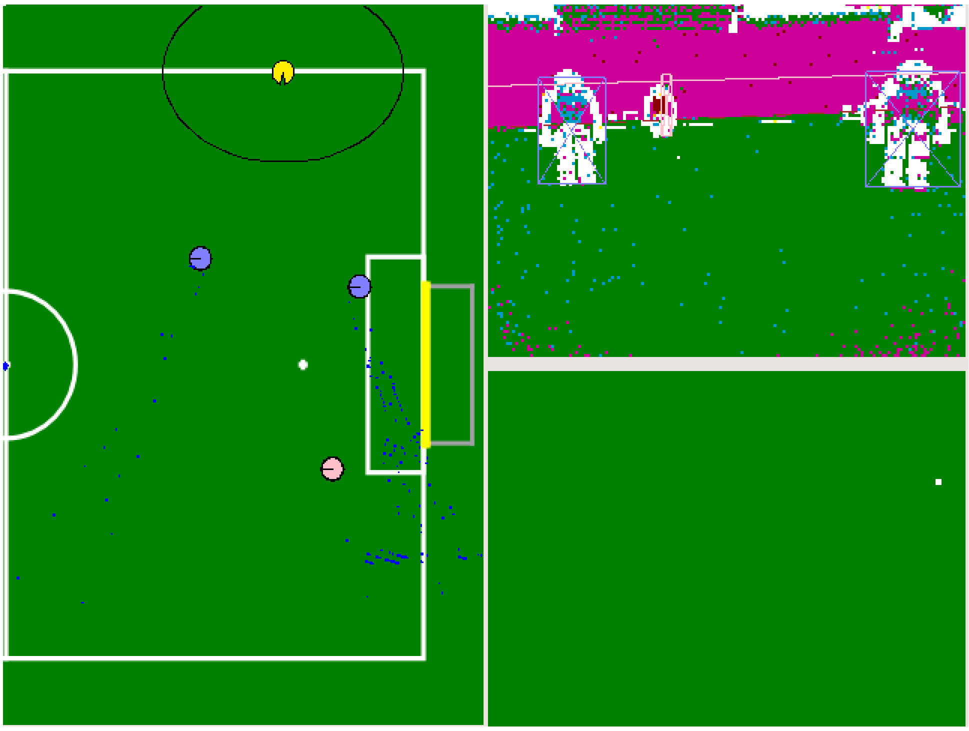

Figure 4.7: Successful robot filter merging observations

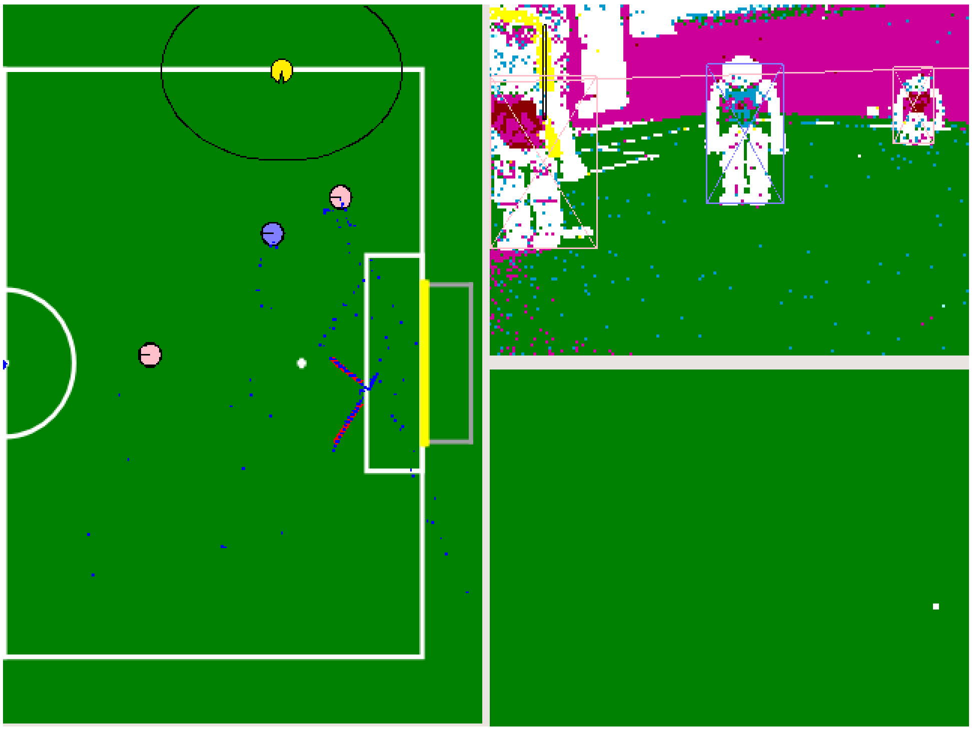

Figure 4.8: Successful robot filter merging observations 2

4.5.2 Competition

The robot filter output was used extensively in the Striker code, which is the behaviour module which determines the behaviour of the robot with the ball. The Striker uses the robots detected through the robot filter to determine what action it wants to do, such as dribble, shoot or shoot at an offset (if there was a goalie in the way). If the robot filter was not successfully working, the striker would be kicking into the opponents or trying to dribble through opponents to get to the goal. The game play seen in the 2014 RoboCup competitions, such as the semi-final against BHuman[4], clearly shows the striker performing well against the opponents. Specifically 2:30 through the game[5], the striker sees an opponent and runs the Ronaldo behaviour whose goal is to dribble around the opponent.

4.6 Evaluation

As can be seen from the test results and the application in the competition, the robot filter was able to successfully decipher and merge robot observations into distinct obstacles. As the striker code was heavily dependent on good robot obstacles coming out of the filter, it is seen that it is working well enough for the robots to avoid or contend with the opponents. This was crucial to rUNSWift winning the competition in 2014 and therefore was a successful implementation.

Chapter 5

Robot Avoidance

5.1 Introduction

Robot avoidance is crucial in the game play as the less the robots run into other robots the fewer penalisations will be applied. This has become more significant in this year's competition due to an increased penalty for pushing (45s), therefore implementing a robust robot avoidance code is crucial to allowing successful game play.

5.2 Goal

There are two main methods of robot avoidance that have been implemented. The first is trying to avoid future robots that are in the path of the robot and the second is emergency avoidance where a robot has come close and it must try to avoid it to stop bumping into it. Some obvious restrictions are that it is hard to detect a robot behind or on the side so avoiding it is implausible. Therefore, reducing the amount of time the robot is walking sideways or backwards is a method to reduce this problem.

5.3 Implementation

5.3.1 Calculating a destination vector

Given a desired destination, the robot needs to be able to calculate the vector that it needs to move on to get there. If there were no obstacles in the way, it would simply be a vector to the destination. However, when objects get in the way, a modified vector needs to be calculated to avoid the robot. In the original code, calculating a vector to a destination or an avoidance vector was repeated throughout the code, which resulted in multiple places having complicated trigonometry. The goal of the refactor was to move the vector calculation to a single point in the code, which would make it easier to understand.

This was achieved by first determining the destination point that the robot desires to go to and then modifying this destination via the far away avoidance and close avoidance functions. At the end of this step, the final point is taken and the vector is calculated. This greatly simplified the code as there was only one point to calculate the vector, instead of three as happened previously.

5.3.2 Far-away-avoidance

Goal

The goal of far-away avoidance is detecting that there is a robot in the way of the target destination and determining a modified vector that will avoid this obstacle. This code is dependent entirely on the visual detection of robots because sonar is not effective enough at far distances. The problem can be seen in Figure 5.1.

Naive implementation

The naive solution to this problem is to find the minimum distance to the left or right of the robot and to go towards that instead of going straight. This is a very simple solution with the result that it eventually reaches a point that is away from the robot. This can be seen in Figure 5.2. The problem with this solution is that the robot eventually overlaps with the minimum safe distance of the other robot. This will result in an emergency avoidance having to be triggered, which is slower then if it just went to the right vector in the first place. The optimal/mathematical solution described next will solve this problem.

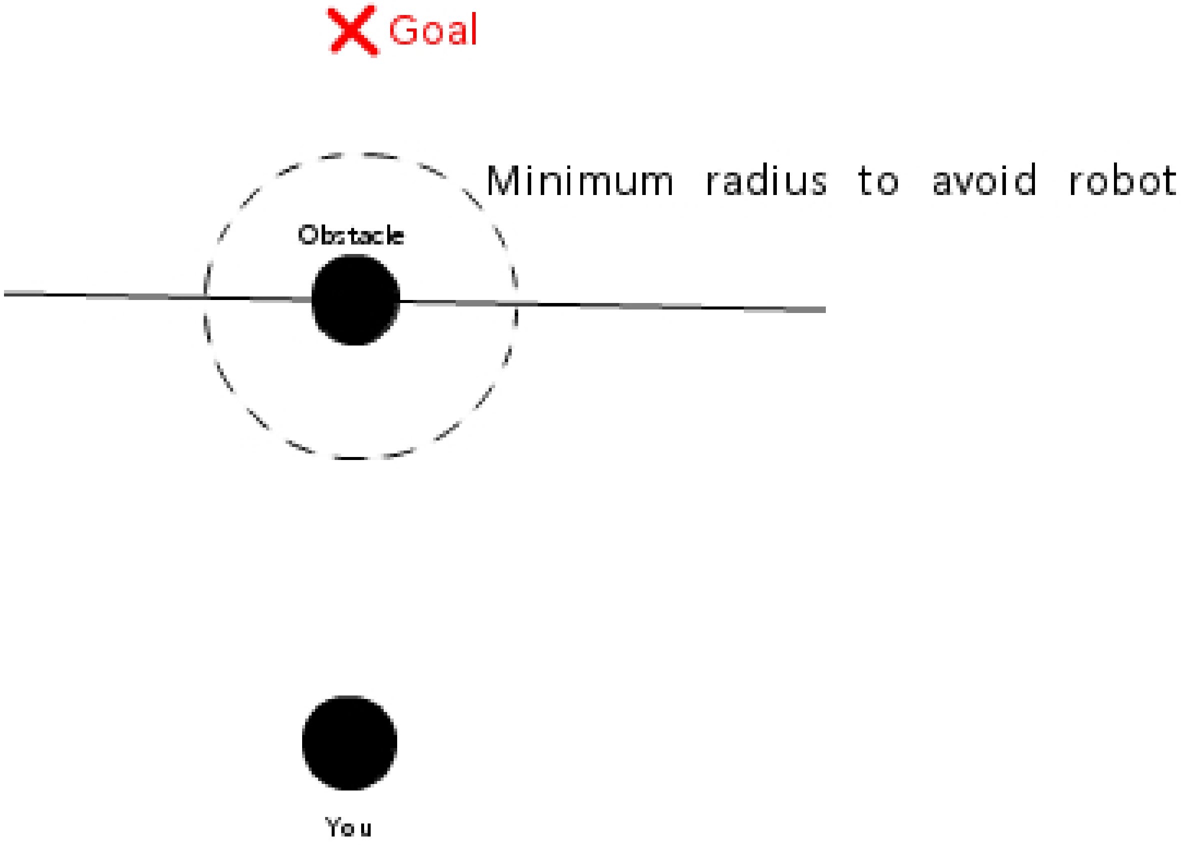

Figure 5.1: Far away obstacle to avoid

Avoiding via tangential vector

The other approach to avoiding the robot is to find a line going from itself that is a tangent to the minimum safe avoidance radius circle around the obstacle. This would result in an optimal path as it is the shortest path that does not overlap with the obstacle. This method can be seen in Figure 5.3.

If this method was to be repeated through the entire walk process, the resulting vector would be seen in Figure 5.4.

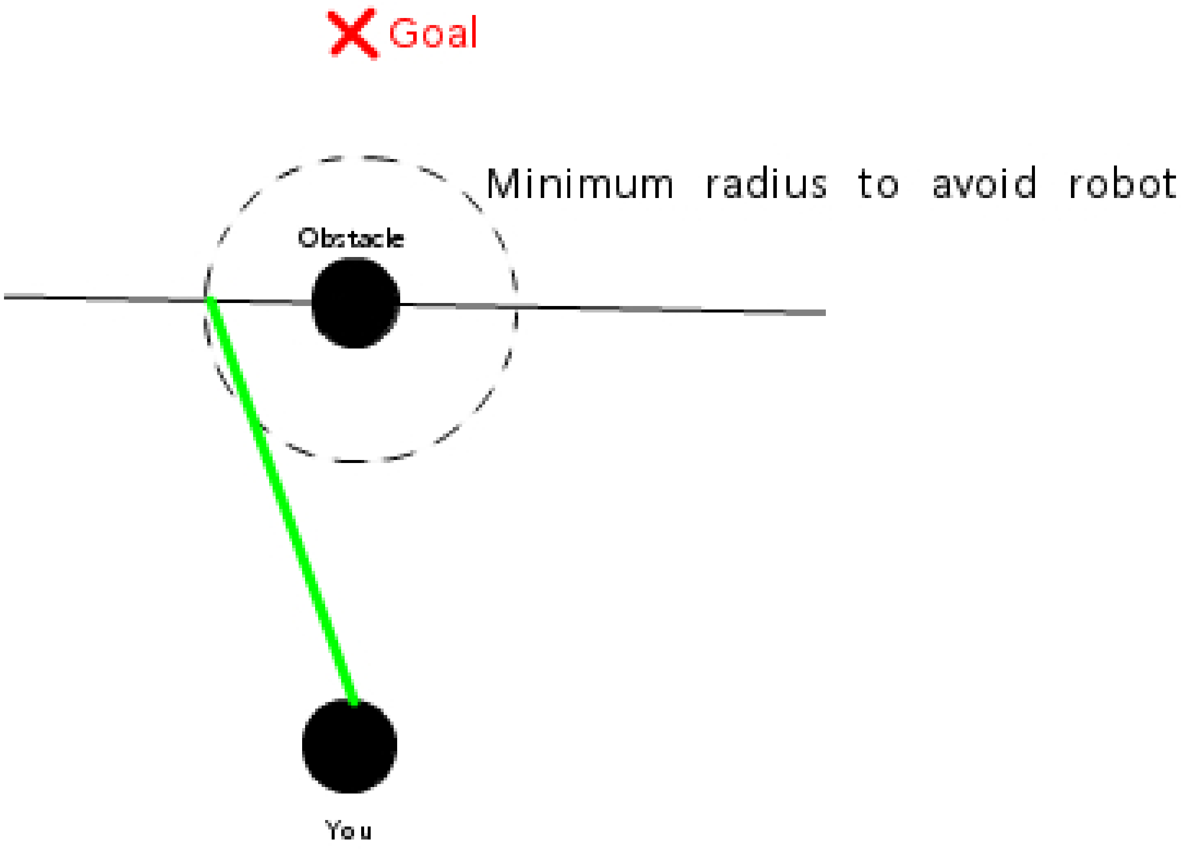

Figure 5.2: Vector to naively avoid the robot

Mathematical implementation

As this is a more complicated solution than the naive solution, the mathematics to calculate the vector is more difficult. The diagrams above have been converted to a mathematical diagram as shown in Figure 5.5.

There is a circle at position (d,0), with radius r, which represents the minimum distance to avoid the robot. The goal of the calculation is to find the length a, and the angle θ to determine the vector that needs to be travelled to avoid the robot. As the lines marked r and a are perpendicular, it can be deduced through Pythagoras that $a = \left|\sqrt{d^2 - r^2}\right|$. This is seen in Equation 5.1.

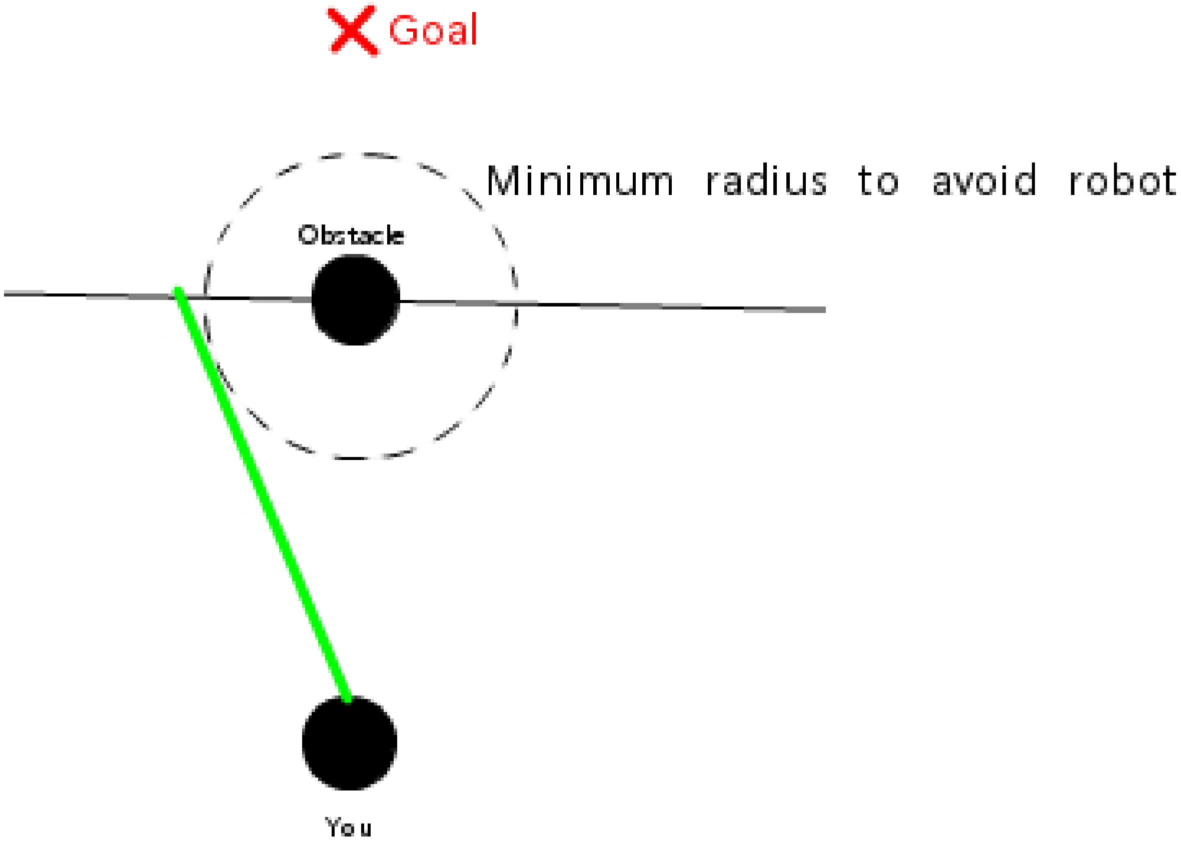

Figure 5.3: Vector to optimally avoid the robot

$$\begin{aligned} a^2 + r^2 &= d^2 \\ \therefore a^2 &= d^2 - r^2 \\ \therefore a &= \left|\sqrt{d^2 - r^2}\right| \end{aligned} \tag{5.1}$$

We can now use trigonometry to calculate the value of θ as shown in Equation 5.2.

$$\begin{aligned} \theta &= tan^{-}1(\frac{r}{a}) \\ \theta &= tan^{-}1(\frac{r}{\left|\sqrt{d^2 - r^2}\right|}) \text{ using Equation 5.1} \end{aligned} \tag{5.2}$$

Therefore the final avoidance vector has distance a and heading $\theta + \beta$.

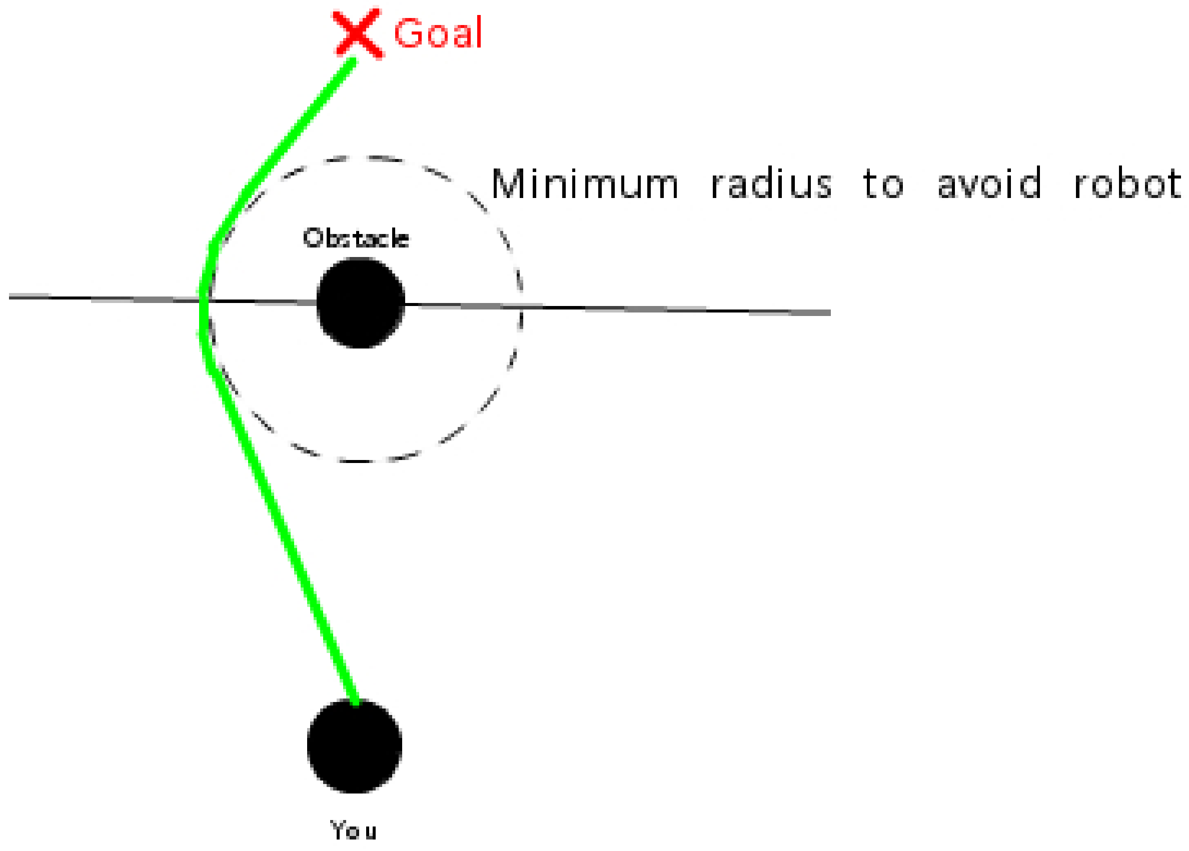

Figure 5.4: Full path to optimally avoid robot

5.3.3 Close robot avoidance

The close robot avoidance is based on the work done by Ritwik Roy who implemented the sonar avoidance. This code was modified to simplify it, as well as applying close emergency visual avoidance.

Goal

The goal of the close robot avoidance is to reduce the possibility of colliding with other robots, which can cause the robot to fall over or to be penalised for pushing.

Sonar Implementation

The sonar avoidance is one of the most important code for the close range robot avoidance. This code was written by Ritwik Roy and worked well in avoiding the close robots.

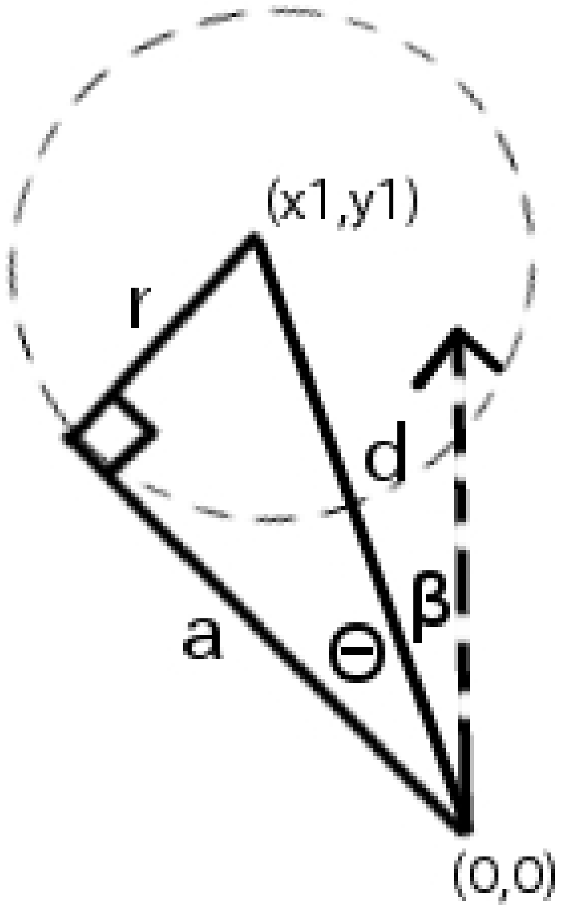

Figure 5.5: Mathematical Diagram showing what needs to be calculated for the vector

This works by using the sonar readings to determine whether there is a robot within a certain distance to the robot. If this distance is too small, therefore potentially causing the robot to run into it, it changes its destination to avoid it. It tries to avoid it by moving perpendicular to the obstacle, e.g. sidestepping left or right. If the object is on the left of the robot, it sidesteps right and vice versa. When implemented into the code, this worked satisfactorily.

With the changed implementation - that is not calculating a vector inside the sonar avoidance code - this was modified by finding a destination point to go to on the left or right of the robot. This could be simply calculated as the point 10cm to the left or right of the robot. It then goes through the vector calculation code that all components would go through to calculate the vector.

5.3.4 Visual Implementation

Sometimes there might not be sonar information or the sonar information might be wrong so a method to avoid close robots effectively is needed. The visual avoidance code described above would not work if it is within the circle, therefore it does something similar to the sonar implementation described in that it tries to sidestep 10cm to the right or left, which ever is better.

5.4 Results

As it is hard to test whether the robot avoidance works well in a controlled environment, a comparison of the robot avoidance between the 2014 and 2013 rUNSWift code has been used. This will be comparing the 2013 Semi Final between rUNSWift and HTWK[3] and the 2014 Semi Final between rUNSWift and BHuman[4]. There are three criteria being measured in each game:

- **Number of contentions**: this is the number of times a rUNSWift robot is competing to get the ball with an opponent. This requires the ball to be near/in-between two opposing robots.
- **Penalised for pushing**: the number of times a rUNSWift robot is penalised for pushing.
- **Falling over from pushing**: itself or a teammate has fallen over because it is pushing another robot.

Some things to consider:

- The contention count has been included because, when the 2013 and 2014 games are compared, the rUNSWift robots are contending for the ball much more often, therefore having a higher chance of bumping into a robot.

- The pushing penalised time was increased between 2013 and 2014 so it is much easier to get penalised for pushing in 2014.

- The success rate is calculated as $\frac{\text{contentions - penalised - fallen}}{\text{contentions}}$

Table 5.1: Comparison of robot avoidance between 2013 and 2014 rUNSWift code.

	2013	2014
Contentions	24	41
Penalised pushing	0	3
Fallen over by running into robot	14	4
Success rate	41.7%	82.9%

Table 5.2 shows the game controller statistics about pushing penalisations for the 2014 RoboCup tournament. This table shows the number of times a team's robot has been penalised for pushing, the number of games, and the average pushes per game. This has been ranked and rUNSWift was ranked 10th, which is on the top half of the table. The mean for the pushes per game rate is 1.878, which is also larger than rUNSWift's rate. Therefore, by comparing rUNSWift to other teams in the league it can be seen that rUNSWift do better then average.

5.5 Evaluation

As can be seen from the data in Table 5.4 the success rate for the 2014 code is almost double the success rate of the 2013 code. This is also significant considering that the 2014 robots were contending almost double the number of times, which is a high point for falling over or getting penalised for pushing. From this, it is evident that the robot avoidance code has been improved significantly.

Table 5.2: Game controller statistics for 2014 matches[7]

Rank	Team	Pushing penalisations	Games	Pushes per game
20	Nao-Team HTWK	38	7	5.43
19	NTU RoboPAL	20	6	3.33
18	TJArk	15	5	3.0
16	Edinferno	11	4	2.75
16	UChile Robotics Team	22	8	2.75
15	Austrian Kangaroos	12	5	2.40
14	Cerberus	9	4	2.25
13	Northern Bites	8	4	2.00
12	UT Austin Villa	7	4	1.75
11	Berlin United	10	6	1.67
10	rUNSWift	11	7	1.57
9	B-Human	9	7	1.29
8	HULKs	5	4	1.25
7	UPennalizers	6	5	1.20
6	MRL-SPL	7	6	1.17
4	Nao Devils Dortmund	5	5	1.00
4	SPQR Team	5	5	1.00
3	RoboCanes	3	4	0.75
1	DAInamite	2	4	0.50
1	Philosopher	2	4	0.50

Chapter 6

Future work

This includes information about future projects that students could complete.

Improved height estimation

At the moment the height of the robot is calculated via trigonometry so does not do well if the robot is crouching or the distance calculation to the robot is not perfect. Therefore a better way to get a true height estimation would help significantly.

Generalise Robot Jersey detection

The jersey detection works by looking for specific jersey colours in the field and is not particularily sophisticated. With RoboCup wanting to move to team specific jerseys, the jersey detection algorithm will need to be modified to cater to all different colours and styles of jerseys. Therefore, a way to improve the jersey detection would be ideal. This detection code was written in its own module so upgrading this would be easy and not impact the other code.

Determine the way the robot is facing

Another good feature from the robot detection code is to determine which way the robot is facing. If it can identify that the robot is facing away it can activate a different task or spend more time getting a good kick compared to if the robot was facing it.

Construct a world model

NOTE: if you are looking for a task relating to vision or localisation you should definitely consider this!

After taking in the robot observations and passing it through the robot filter, it would be good to share this information with the other teammates. We could therefore share information that one robot has detected one that another robot cannot see. It could also help with localisation when one robot sees a teammate on the other end of the field, it could tell them that they are flipped if they think they are on the wrong side.

Better colour calibration tools

The current colour calibration tool is a point and click exercise. We take a frame, as seen in Figure 6.1, and click the pixels that we think should be a specific colour. In this case, we are selecting the pixels that should be blue. When a pixel is selected it also assigns colours that are similar to this colour so we are not having to select every pixel in the frame but we still need to select I high number of them.

One possible way to improve this method is an area selection process. We could select an area and say that these pixels are the field and this area is lines and have an algorithm distribute the colours so that most of the pixels in the area are the correct colour. This is shown in Figure 6.2 where rough boxes around some areas are shown. Note that this has been specifically made rough because it should be an easy process to quickly make areas for the algorithm to run. After this, if you wish to fine tune, you could move back to the single pixel clicking.

Automatic colour calibration and camera settings

One of the problems with a lot of the current code is the need for a good colour calibration. Vision components such as ball detection, goal detection and robot detection use colours and the better the calibration the better these algorithms work.

The process of calibrating for a given field and lighting condition takes a long time, so

Figure 6.1: Zoomed frame that colour calibration is being applied on

a process to automate this or at least a way to get a good colour calibration foundation would save time. Due to natural lighting conditions in the 2014 RoboCup competition, a new colour calibration was needing to be done before each match and then confirmed minutes before the match started. The reason for this is that a cloud blocking the sun can significantly impact the robot's ability to see the ball. This was seen significantly in the first pool match against HULK[6], where a cloud came in halfway through the first half resulting in the rUNSWift robots not being able to see the ball. This was clarified with a quick colour calibration modification at half time.

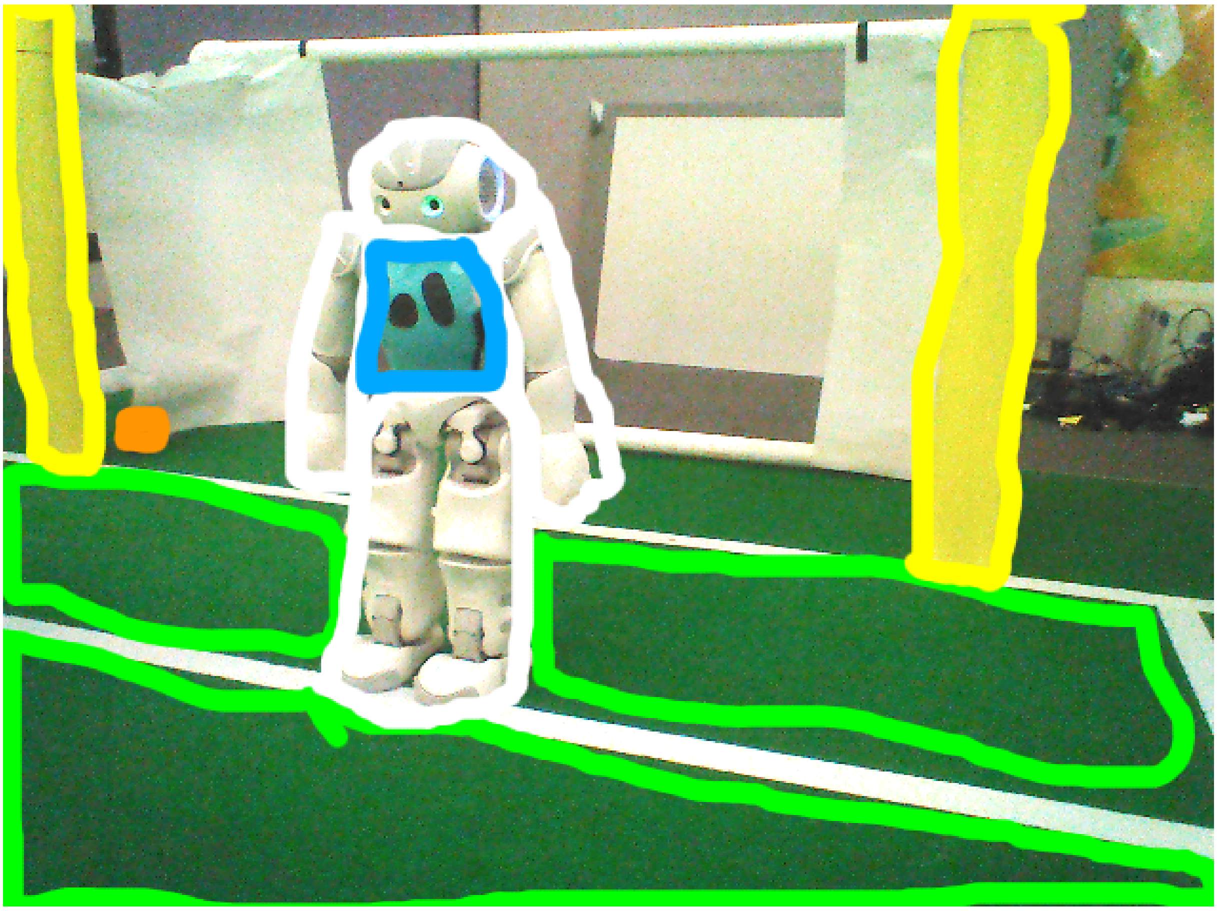

Figure 6.2: Colour calibration via boxes

Chapter 7

Conclusions

In completing this thesis, there have been three main improvements to the rUNSWift RoboCup code base. The robot detection code was rewritten to improve the performance; the robot filter was rewritten to better merge observations into obstacle; and the robot avoidance was improved to better avoid robots. In all of these aspects, there were significant improvements as shown throughout this report.

7.1 Summary

- **Robot detection**: the robot detection code was significantly improved over the old code. The detection success rate in some sample tests jumped from 64% in the old robot detection to 87% in the new. The significant improvements were in the better detection for far away objects as well as improved close range robot detection through extensive use of the bottom camera.

- **Robot filter**: when applying the new robot detection into the robot filter it was seen to perform poorly. This was rewritten to improve the merging ability and the result was far more stable obstacles. The application of the robot filter is seen through the Striker behaviour in the competition matches, where the robot was able to see and avoid or contend with opponent robots.

- **Robot avoidance**: the new robot avoidance utilized the visual observations from the robot detection as well as utilising the sonar information. This allowed it to make better decisions to get around a future opponent as well as quick changes in direction to avoid a close robot. Comparing the robot avoidance in rUNSWift's 2014 code to the previous year's , along with other teams in the 2014 competition, it can be seen that the 2014 code performs particularily well.

7.2 Closing

With improvements in all aspects of robot detection and avoidance, we were able to see the robots improve considerably in the overall game play in the 2014 competition. Additionally, being able to reliably detect and avoid robots reduced the number of times the robot fell over. Also the increased knowledge about opponent's position on the field, the behaviour code, especially the Striker, is able to make informed decisions on the best way to beat the opponent more times than not. Throughout the competition, the rUNSWift robots were seen to beat the opponents countless times and, to a considerable degree, this is thanks to the robot detection that the Striker can utilise. While there are many are in which the robot detection, filter or avoidance algorithms could be improved, this helped considerably in improving the code base so as to make UNSW number 1 in the world in 2014. Hopefully, it will also help maintain this ranking in the future.

Bibliography

[1] Jimmy Kurniawan *Multi-modal Machine-learned Robot Detection for RoboCup SPL* 2011.

[2] BHuman 2013 Code Release
http://www.b-human.de/downloads/publications/2013/CodeRelease2013.pdf

[3] 2013 Semi Final rUNSWift vs HTWK *https://www.youtube.com/watch?v=hFTslT9CdgU*

[4] 2014 Semi Final rUNSWift vs BHuman *https://www.youtube.com/watch?v=xlS5D8qMSHE*

[5] 2014 Semi Final rUNSWift vs BHuman - Ronaldo striker play *https://www.youtube.com/watch?feature=player_detailpage&v=xlS5D8qMSHE#t=141*

[6] 2014 Pool match rUNSWift vs HULK *https://www.youtube.com/watch?v=y_s0TDn_CSs*

[7] 2014 RoboCup Game Controller Statistics *http://www.informatik.uni-bremen.de/spl/pub/Website/Results2014/RoboCup2014Statistics.pdf*