

# The 2005 NUbots Team Report

Michael J. Quinlan      Steven P. Nicklin      Kenny Hong  
Naomi Henderson      Stephen R. Young      Timothy G. Moore  
Robin Fisher      Phavanna Douangboupha      Stephan K. Chalup  
Richard H. Middleton      Robert King

February 2, 2006



School of Electrical Engineering & Computer Science  
The University of Newcastle, Callaghan 2308, Australia  
<http://www.robots.newcastle.edu.au>

## Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>System Architecture</b>	<b>6</b>
2.1	2005 Design . . . . .	6
2.2	Python . . . . .	7
<b>3</b>	<b>Vision System</b>	<b>8</b>
3.1	Overview . . . . .	8
3.2	Soft Colour Classification . . . . .	9
3.2.1	Ball Classification . . . . .	9
3.2.2	Yellow Goal and Yellow Beacon Conditions . . . . .	12
3.2.3	Yellow Goal and Yellow Beacon Conditions . . . . .	14
3.2.4	Pink Beacon Conditions . . . . .	15
3.2.5	Red Robot Conditions . . . . .	15
3.2.6	Blue Robot Conditions . . . . .	16
3.3	Ball Detection . . . . .	17
3.3.1	Ball Distance, Bearing and Elevation Calculations . . . . .	17
3.3.2	Height Check . . . . .	18
3.3.3	Surrounding Colour Test . . . . .	19
3.3.4	Circle Fitting . . . . .	20
3.3.5	Image Skew Correction . . . . .	22
3.4	Goal Detection . . . . .	27
3.4.1	Goal Candidate Construction . . . . .	27
3.4.2	Goal Candidate Cut-Off Detection . . . . .	28
3.4.3	Ball Distance, Bearing and Elevation Calculations . . . . .	28
3.4.4	Goal Identification . . . . .	30
3.4.5	Goal Post Detection . . . . .	30
3.4.6	Goal “Gap” Detection . . . . .	32
3.5	Blob Rotation . . . . .	33
3.6	Filtering Algorithm . . . . .	35
3.7	Beacon Detection . . . . .	36
3.7.1	Beacon Distance, Bearing and Elevation Calculations . . . . .	37
3.7.2	Non-ideal beacon blobs . . . . .	38

3.8	Robot Detection . . . . .	38
3.8.1	Robot Clusters and Containers . . . . .	39
3.8.2	Orientation Recognition . . . . .	40
3.8.3	Red Robots . . . . .	41
3.8.4	Blue Robots . . . . .	41
3.9	Line Detection . . . . .	45
3.9.1	Point detection . . . . .	47
3.9.2	Line creation . . . . .	47
3.9.3	Corner detection . . . . .	49
3.10	Identifying potential obstacles using scan lines . . . . .	50
<b>4</b>	<b>Localisation and World Modeling</b>	<b>51</b>
4.1	Kalman Filter . . . . .	51
4.2	Time Update . . . . .	53
4.3	Measurement Update . . . . .	54
4.4	Filter Reset . . . . .	55
4.5	Clipping . . . . .	56
4.6	Shared Ball . . . . .	57
<b>5</b>	<b>Locomotion</b>	<b>58</b>
5.1	Walk Engine . . . . .	58
5.1.1	Individual Leg Control . . . . .	58
5.2	Head Control . . . . .	59
5.2.1	General movements . . . . .	59
5.2.2	Tracking an object . . . . .	60
5.3	Forward Kinematics . . . . .	61
<b>6</b>	<b>Team Behaviour</b>	<b>63</b>
6.1	Positions / Roles . . . . .	63
6.1.1	Formations . . . . .	63
6.1.2	Positions . . . . .	64
6.2	Potential Fields . . . . .	64
6.2.1	Points . . . . .	65
6.2.2	Lines . . . . .	66

<b>7</b>	<b>Individual Behaviours</b>	<b>68</b>
7.1	Chase . . . . .	68
7.1.1	‘Avoiding’ a robot while chasing . . . . .	70
7.2	Grab . . . . .	70
7.2.1	Grab Trigger . . . . .	71
7.2.2	Grab Trigger - Learning . . . . .	71
7.2.3	Grab Motion . . . . .	72
7.3	Dribbling (“Holding”) . . . . .	73
7.3.1	Lining up to shoot at goal . . . . .	76
7.3.2	Obstacle Avoidance - “Vision” . . . . .	77
7.3.3	Obstacle Avoidance - “Infrared” . . . . .	79
7.3.4	Avoiding sidelines . . . . .	80
7.4	Kick Selection . . . . .	81
7.5	Moving to a Point . . . . .	82
7.6	Diving . . . . .	82
7.7	Unused Behaviours . . . . .	83
7.7.1	“Paw” Dribbling . . . . .	83
7.7.2	Controlled Diving . . . . .	83
<b>8</b>	<b>Debugging and Offline Tools</b>	<b>84</b>
8.1	Simulator . . . . .	84
8.1.1	Experiences . . . . .	84
<b>9</b>	<b>Challenges</b>	<b>86</b>
9.1	Open challenge . . . . .	86
9.2	Variable lighting challenge . . . . .	87
9.3	Almost SLAM . . . . .	88

## 1 Introduction

Fresh from three consecutive 3rd place finishes at Robocup 2002, 2003 and 2004 the NUbots in 2005 faced arguably their most challenging and exiting year. 2005 saw our largest intake of new undergraduate developers and simultaneously saw the exit of one of the original 2002 developers. This large scale personnel change was foreseen and was the motivation for the following statement -

*“.. the NUbots are intending to perform a complete rewrite for 2005.”*

- The 2004 NUbots Team Report

By October 2004 the NUBot code base was officially reduced to 0 lines of code and the initial preparation for the new students began. The 2004 members constructed a ‘base’ system that contained only an embedded Python interpreter and the basic interfaces to the hardware (connections to the camera, sensors etc). To encourage fresh approaches the new students would not be given source code from any previous NUBot teams. Initially the rewrite gave the impression of a slow development cycle, the first time we had four robots booted and communicating was one-day before the Australian Open, but the rewrite forced the new developers to fully understand the problems faced in this league. In the long run this increased understanding was an essential ingredient in the rapid yet high quality development of the code which would follow.

Although the state of the rewrite was a cause for concern at times, the potential of the code was always evident. Our 2nd place finish at Robocup is a clear indication that substantial code can be developed every year. The 2005 code will provide the base for 2006 code, but it is likely that another rewrite will take place in 2007.

Robocup 2005 was by far our most successful Robocup, finishing second in both the soccer competition and the challenges. The second place in the soccer was particularly important as it marks our first non-third finish. We are extremely proud of the quality of both the grandfinal (German Team) and semi-final (rUNSWift) matches. The challenge result was also exciting, this was the first time since 2002 that we had placed in the top three.

## 2 System Architecture

The basic design is similar to that employed in previous years, that is one control module *NUbot* and four functional modules - *Vision*, *Localisation & World Modeling*, *Behaviour* and *Locomotion*. The most substantial change was the addition of the embedded Python interpreter (Section 2.2).

### 2.1 2005 Design

The flow of information in our system can be seen in Figure 1. It has been observed that in every year since 2002 the code has been migrating towards a single global store of variables (approaching a whiteboard architecture), these variables can then be written to and read by each module as required. In the past we have run into problems where new ideas require variables that were private in other modules, often it is not trivial to retrospectively make these variables public or to pass them around. The complete rewrite enabled us to setup the principle of the global store, in addition we have also forgone the use of private variables in all classes. We acknowledge that these approaches go against many of the established software engineering design principles but we feel they are well suited to the agile programming (rapid prototyping) development employed in Robocup.

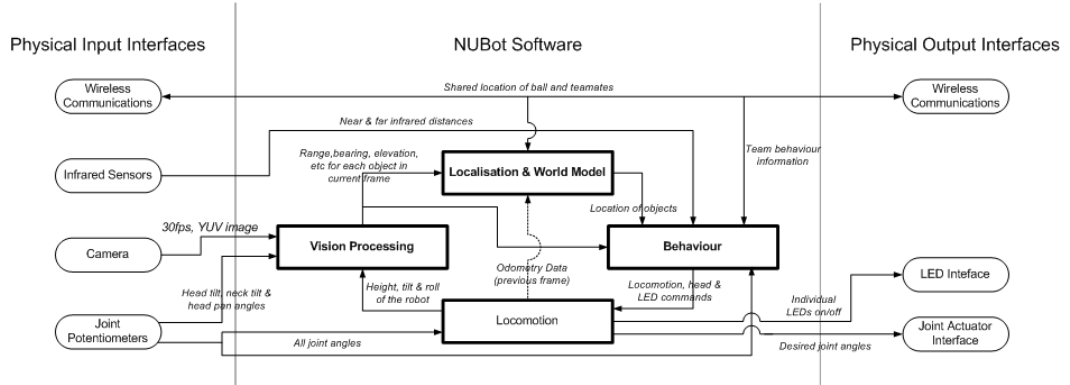


Figure 1: 2005 Software Architecture

## 2.2 Python

Although the use of a scripting language in Robocup was not new to us (Melbourne 2002, UPenn 2003→), the idea of using Python was suggest by rUNSWift at the completion of Robocup 2004. Upon a brief investigation into the Python language we felt that the idea warranted some on robot testing. Cross-compiling the Python library to the robot was not trivial, but once done the benefits became clear. The ability to modify code or write entire classes/methods without rebooting the robot **greatly** reduces development time.

At one stage we had blob formation in C++ then the remaining vision, localisation, behaviour and locomotion all in Python. Unfortunately we began to run into speed problems, as a result we started moving computationally expensive code back to C++. By Robocup, Python was primarily used for behaviour, this module never contained C++ code. Additionally robot recognition, some parts of ball detection and the control of vision and locomotion were implemented in Python. For the remainder of this document we assume code is implemented in C++ unless specified otherwise.

In hindsight many of the speed problems can be attributed to our lack of Python experience, for 2006 we hope to move much of the C++ back to Python.

### 3 Vision System

The major developers in the vision module were all new to the team and they followed the instructions not to look any previous code, they did however consult previous team members so a certain level of similarity to previous years does exist. The major change to the 2005 vision system was the introduction of ‘soft colours’ (3.2). Soft colours increased the information that could be used in blob formation and object recognition, so substantial modifications to these rules were needed.

Robot detection and line detection were completely re-done and the methods hold no similarity to previous NUBot attempts.

#### 3.1 Overview

Vision follows a similar execution path as in previous years : *colour classification, blob formation and object detection* (Figure 2). Colour classification and blob formation are relatively common principles amongst Robocup teams, for further information read our previous reports [Chalup *et al.*, 2002; Bunting *et al.*, 2003; Quinlan *et al.*, 2004]. It should be noted that unlike other teams [Röfer *et al.*, 2004] we blob form on every pixel. This enables us to form blobs as small as 1 pixel. Although in practice we never use blobs with an area less then 4 pixels.

The main new features in this year’s vision are: soft colour, secondary colour classification with edge detection, line fit to posts, ball skew correction, blob rotation, blob filtering, robot recognition and line detection. Further information on most of the new features can be found in [Henderson, 2005; Hong, 2005; Nicklin, 2005].

One subtle change this year was the replacement of the 6-bit lookup table (LUT) with a new 7-bit LUT. The new LUT allowed for better separation of colours but resulted in 2MB table, to overcome this we used gzip compression on the robot to compress/uncompress the LUT to about 12k.

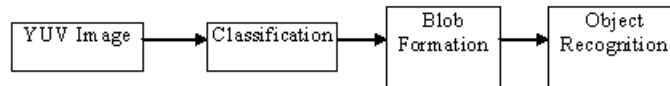


Figure 2: NUBot vision system. Note: Line detection is not shown in this diagram but exists parallel to blob formation and object recognition



### 3.2 Soft Colour Classification

Soft colours are additional classified colours that act as a buffer between object colours. They are needed because in different lighting conditions object colours share YUV values. For example shades of the ball may overlap with shades of the red uniforms when a shadow is cast - Red Orange buffers this overlap. The soft colours in the NUbots system are : Shadow Blue, Red Orange, Yellow Orange, Pink Orange, Ball Orange and Shadow Object.

Blob Formation creates blobs of classified colours- object blobs and soft blobs, which are then filtered and modified in the Soft Colour Filtering Process. Soft colour blobs are conditioned against corresponding object colour blobs and if conditions passed then either directly passed through to object recognition, or blob size modified and then passed on, or associated via object array and passed through to object recognition

Each object on the field has particular blob formations depending on lighting, shadowing, angle and distance. These characteristics were thoroughly studied and tested using collections of image streams to create conditions for the soft colour filter software

#### 3.2.1 Ball Classification

The ball is classified as shown in Figure 3. Red orange is classified to be that of the shadow, orange is classified to be the main area of the ball, yellow orange is that of the light patch. Examples of blob formation using this classification system are shown in Figures 4 - 7. Through study of blob formation conditions for each soft colour were determined.

##### Red Orange Conditions

A red-orange blob must overlap an orange blob to be part of the ball. The red-orange blob may be wider than the orange blob, narrower, offset or enclosing it as shown in Figure 8. However the as red orange classifies the shadow of the ball a red orange blob cannot be an upper overlap i.e. the maximum-y of the red-orange blob cannot be above the maximum-y value of the orange blob, as shown in Figure 9.

If there is no orange in the image yet a large red orange blob exists, it is likely to be the ball at a close proximity with shadow cast by the robot. Large red orange blobs are passed in the initial size checks and do not need to be conditioned against

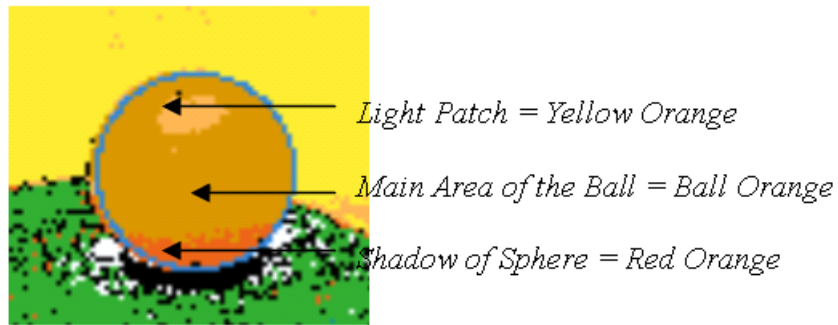


Figure 3: Ball classification principal.

orange.

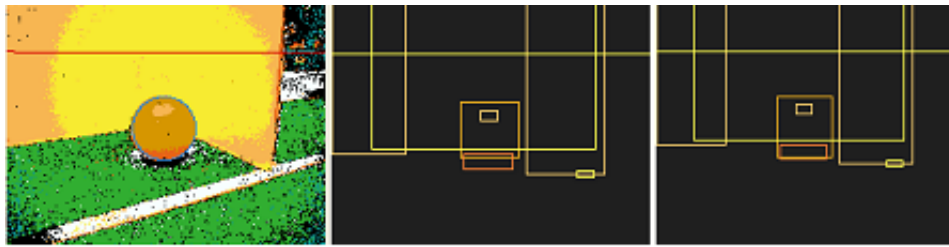


Figure 4: Ideal lighting of ball, its blob formation and its filtered size modified blob formation.

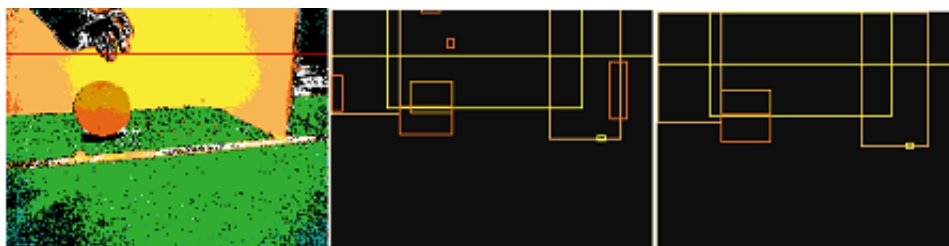


Figure 5: Shadowing of ball, unnecessary red orange blob formation, filtered size modified blob formation.

Each red-orange blob that is passed (has acceptable overlap) is then compared to the associated orange blob. Comparison is of minimum and maximum values and the orange blob modified to fully enclose the red-orange blob.



Figure 6: Corner images have increased noise, shadow creates unwanted red orange blobs, these are filtered out and the size of the orange blob is modified to the maximum dimensions of the overlapping red orange blob.

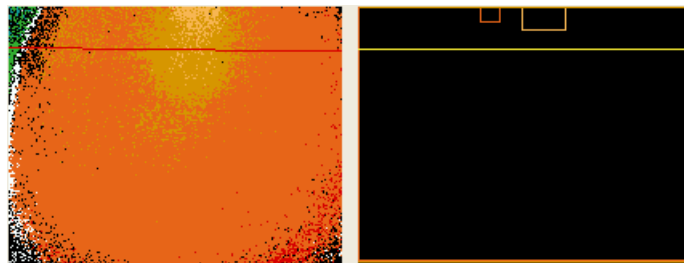


Figure 7: Close image of ball, the orange blob parameters are within the red-orange blob, the orange blob parameters modified to the most minimum and maximum of the two, in this case the size of the red-orange blob.



Figure 8: Acceptable red-orange / orange blob formation.

### Yellow Orange Light Patch Conditions

The light patch must be within an orange blob or overlap the top of an orange blob, but must not be wider as shown in Figure 10.



Figure 9: Unnecessary blob formation.



Figure 10: Light patch blob formation.

### 3.2.2 Yellow Goal and Yellow Beacon Conditions

The goal and beacons are classified to be beacon yellow for the main area under good lighting conditions. The darker shades of beacon yellow are classified yellow orange and tend to be the corners of the goal posts and edges of close images as shown in Figure 11.

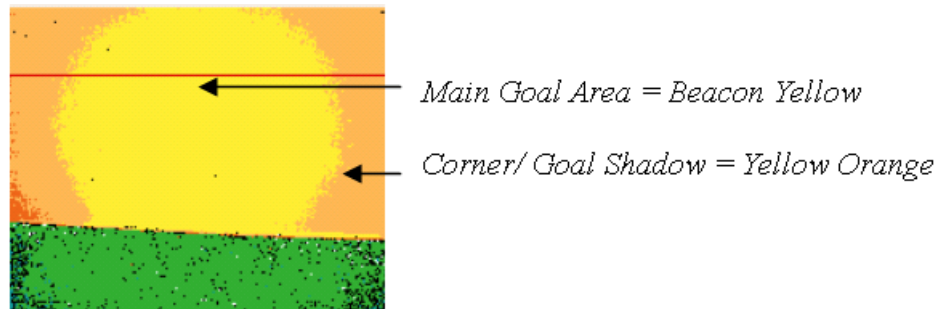


Figure 11: Yellow Goal Classification Principal.

By studying blob formation of goals it was noted that any overlap patterns between the object blob and soft blob were acceptable. For goals and beacons the conditions are: if the compare blob is not inside or outside the test blob then it must be overlapping.

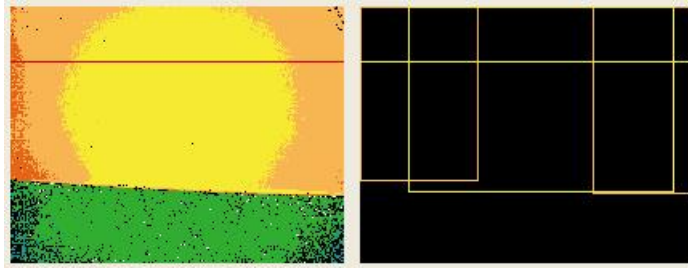


Figure 12: Classified image of goal and blob formation. Information of goal width is maintained without over-classification.

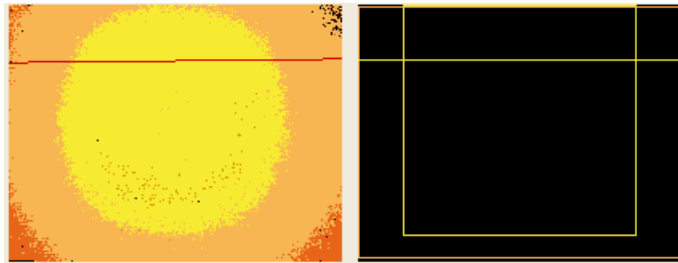


Figure 13: Image of close goal and blob formation.

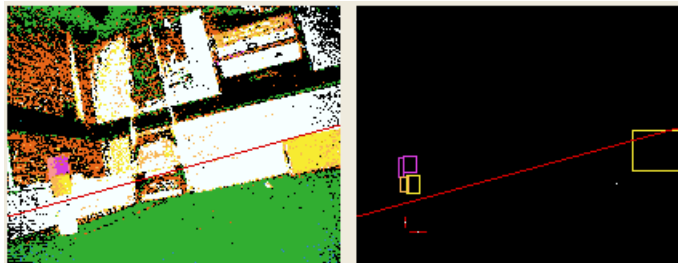


Figure 14: Image of beacon and soft colour blob formation.



Figure 15: Acceptable blob formations for goals and beacons.

### 3.2.3 Yellow Goal and Yellow Beacon Conditions

The blue goal and beacons are classified to be beacon blue and darker shades (typically the edges of the object and corners of the goal) as shown in Figure 16.

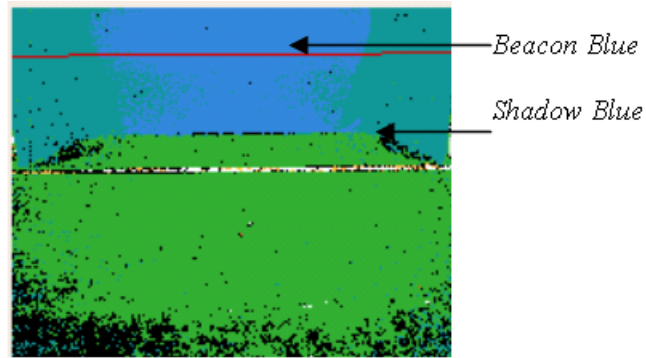


Figure 16: Classification Principal of Blue Goal.

For acceptable blob formations of blue goals and beacons see Figure 15. Additional filtering is required to remove inner blue blobs characteristic to blue shadow classification as seen in Figure 17.

Large shadow blue blobs located at the sides of an image have the possibility of being a goal corner and are passed in the initial size checks therefore do not need to be conditioned against an object blob (Figure 18).

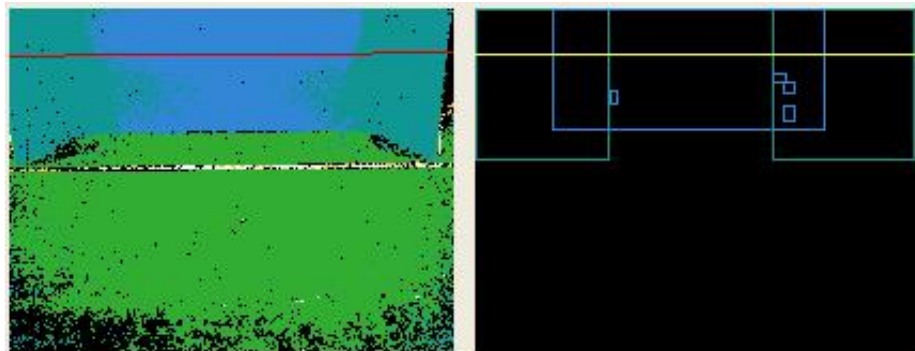


Figure 17: Blue blobs formed inside blue goal blob due to dispersion of shadow blue classification.

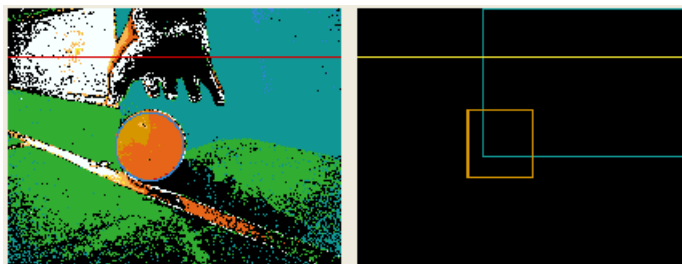


Figure 18: Corners of blue goal are very dark and classified as shadow blue. Occasionally there is no blue seen in the image, for this reason shadow blue blobs of considerable size located along the edge of an image are passed.

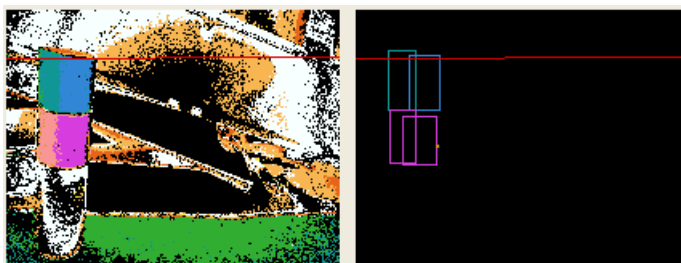


Figure 19: Blob formation of 3-Dimensional beacon characteristic.

### 3.2.4 Pink Beacon Conditions

The blob formation of pink-orange and beacon-pink can be seen in Figure 19 and the logic for the filter seen in Figure 15.

Beacon pink is also overlap checked against yellow orange for the case that the lighting of a beacon is insufficient to make see it as yellow. The yellow orange blobs passed are used in a secondary beacon check after yellow.

### 3.2.5 Red Robot Conditions

Robot red is classified to be that of the red uniform and any overlap between red uniform shades and ball shadow shades are classified to be red orange.

For red orange to be associated with robot red it must overlap. Any overlap of robot red and red orange results in the expansion of maximum and minimum parameters for the robot red blob. After conditioning of blobs and expansion, and

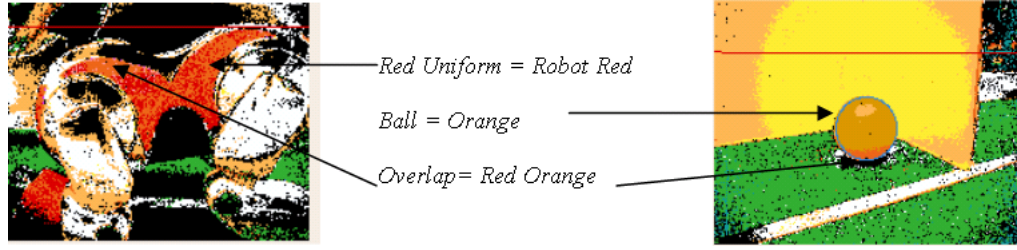


Figure 20: Red / Red orange Classification principal of red robot and shades shared with ball.

also, due to the nature of scattered blob formations (similar to that of Figure 21) overlap occurs between two expanded red blobs. Any secondary overlap is also acceptable between robot red and robot red and expansion occurs.

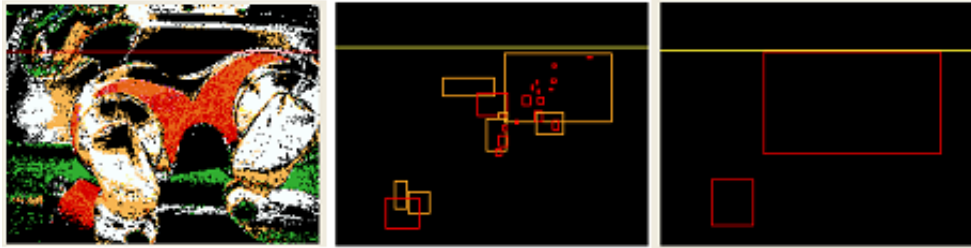


Figure 21: Red and red-orange are dispersed creating multiple blobs. Red orange overlapping red are ignored but red parameters expanded to account for actual size.

### 3.2.6 Blue Robot Conditions

One method of blue robot blob formation was to classify the entire spectrum of dark shades to be blue and then pass through initially based on size and later eliminated based on sanity checks.

However, with the introduction of secondary colour classification using edge detection for robot blue this method is not required. The robot blue blobs formed essentially are already conditioned and only need to be sanity checked in object recognition for cases where a non-blue robot pixel arrangement passed the secondary classification conditions.



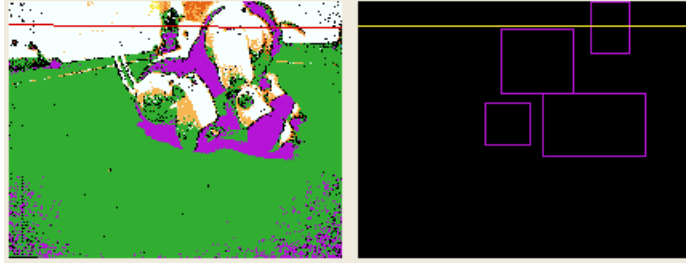


Figure 22: Dark shades are classified as object-shadow, see section 3.8.4 for conversion to robot blue using edge detection.

### 3.3 Ball Detection

The elements that are examined before a blob is identified as a ball include the blobs correct pixel ratio, height above the ground, the colour below the ball, and also its roundness. A confidence system is used to bring all of these elements together to come to a final decision. The ball recognition code was created in Python primarily for the reason that code can be changed on-the-fly and uploaded to the robots, decreasing the required testing time.

#### 3.3.1 Ball Distance, Bearing and Elevation Calculations

The raw values of bearing and elevation are calculated using the centre x and centre y of the blob in the image, while the raw distance is calculated using the width of the blob in pixels. These values are the values relative to the camera. They are then converted using the transform position function. Circle fitting is applied to a ball to improve the accuracy of all three values. When an acceptable circle is found these values are re-calculated using the new centre x and centre y of the circle to find the raw bearing and elevation, while the circles diameter is used to find the raw distance.

The distance to the ball is calculated using the balls physical diameter and the diameter of the ball in pixels. The ‘Effective Camera Distance in Pixels’ is a value calculated using trigonometry and the field of view of the camera along with its resolution. This value represents the effective distance from the camera to the image in pixels. Its calculation is shown in more detail in section 3.5.3. This can then be used as a ratio to convert the diameter of the ball in pixels and the physical

diameter of the ball into the physical distance of the ball as seen below

$$distance(cm) = \frac{Effective\ camera\ distance(pixels) \times Ball\ diameter(cm)}{Vision\ diameter\ of\ ball(pixels)}$$

### 3.3.2 Height Check

The ball rarely leaves the ground during play. Because of this the perceived ball height should be relatively constant. This can be used to identify balls by comparing the calculated height to the expected height. This test relies on both an accurate distance and elevation; therefore any errors in either of these values can cause errors in the height. For this reason a range of heights must be allowed to cater for the noise in the distance and bearing calculations.

The ball rarely leaves the ground during play. Because of this the perceived ball height should be relatively constant. This can be used to identify balls by comparing the calculated height to the expected height. This test relies on both an accurate distance and elevation; therefore any errors in either of these values can cause errors in the height. For this reason a range of heights must be allowed to cater for the noise in the distance and bearing calculations.

The ball height is calculated as the z co-ordinate in the robots 3D space. The z-axis goes from the ground upwards in the positive direction. This z-value is calculated by using simple trigonometry multiplying the radial distance  $d$  of the object by the sine of its elevation  $\theta$  i.e.  $z = d \sin(\theta)$  as seen in Figure 23. This z-value is relative to centre of the robot, and as such should be below zero.

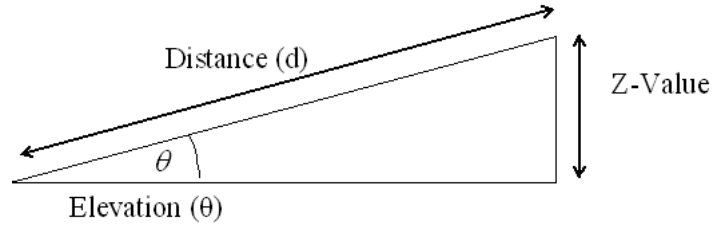


Figure 23: Z-value calculation.

### 3.3.3 Surrounding Colour Test

As previously stated the ball rarely leaves the field during play and most false objects are located outside of the field. From these assumptions testing the colour directly below a blob can be used to determine if an object is on, or off of the field. If the area directly below the blob is field green, then this increases our confidence that the object is located on the field.

It is expected that a ball by itself would be sitting directly above the green of the field. However the robots camera is constantly moving and so the bottom of the image is not always downwards. Therefore the area in which the colour is checked must be rotated so that it does appear below the object. This required a rotation of the co-ordinates of the pixel before the test is done. The following formulas were used

$$x_{rotated} = (x - x_c) \cos \Theta - (y - y_c) \sin \Theta + x_c$$

$$y_{rotated} = (x - x_c) \sin \Theta - (y - y_c) \cos \Theta + y_c$$

Where  $\Theta$  is the angle of the rotation to be performed,  $(x, y)$  is the original point and  $(x_c, y_c)$  is the point about which the rotation is to occur. In this case the rotation occurs about the centre of the blob currently being processed. The theta value used is obtained by finding the angle between the x-axis of the image and a plane parallel to the ground represented by the horizon line that is calculated from the robots current joint positions, see section [Hong, 2005], [Seysener, 2003] or [Röfer *et al.*, 2004] for more detail. Upon implementation of these equations a problem was found whereby once an area had been rotated it may be partially or in some cases fully outside of the image. This required in special cases, where all of this area is located outside of the image that the colour check simply ignore this test. If this is not the case then the number of correctly scanned pixels (i.e. contained inside of the image) is counted as well as those that are found to be of the correct colour. The results are returned as a percentage ratio i.e

$$Correct\ pixel\ ratio = \frac{Correctly\ coloured\ pixels}{Correctly\ scanned\ pixels}$$

This method greatly improved the accuracy of the scans as the pixels that were rotated outside of the image were originally included in the total of scanned pixels. The nature of this test means that it should be able to aid the identification of

the ball however should not be able to on its own reject a blob as the ball. This is because of the many scenarios where the green below the ball may not be seen either because it is blocked by another object or it may not even exist if the ball is sitting on a white field line.



Figure 24: Rotated scan area used for ball recognition (shown in blue).

### 3.3.4 Circle Fitting

Least squares circle fitting has been implemented to improve the distances, bearings and elevations of balls that are not fully visible in the image. The functions used to implement the least squares circle fitting function in the previous years worked well and so they were not re-written. For more information on the least squares fitting function used see [Seysener, 2003],[Seysener *et al.*, 2004]. The function to gather the points on the outside of the ball was also re-written in order to work with the new code structure. These points are gathered during one of various scan types depending on the parts of the ball that are cut-off. Rules to select the scanning direction to use then had to be implemented. The scans work by searching from the outside of the blob inwards until it finds pixels of the same colour of the blob, or in the case of the orange blobs also one of the soft colours close to orange. Soft colours are colours that can belong to more than one “basic” colour (see section 3.2). The directions for which scans have been created are: *Left to Right*, *Right to Left*, *Top to Bottom*, *Bottom to Top*, *Simultaneous Left to centre & Right to centre* and *Simultaneous Top to Centre & Bottom to Centre*. These can be seen in Figure 26.

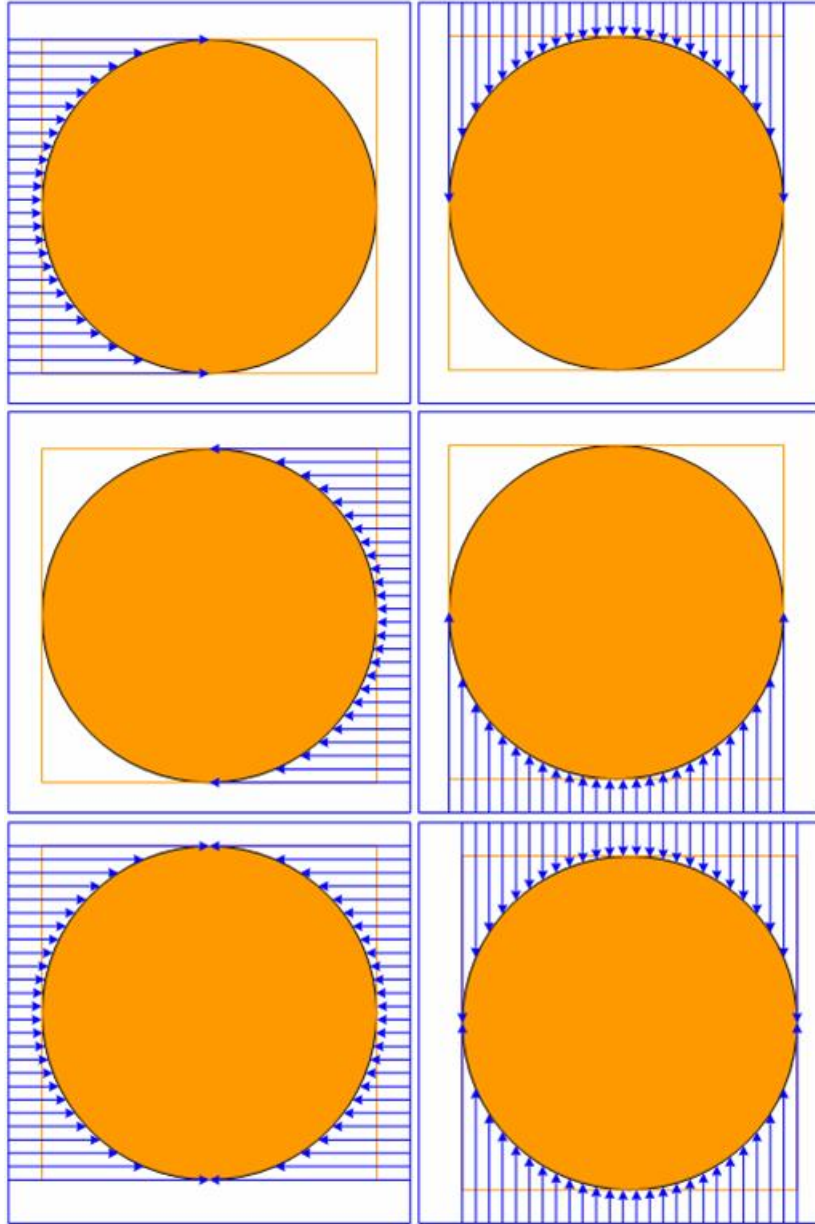


Figure 25: Ball scanning directions, (top left) left to right, (middle left) right to left, (bottom left) left and right to centre, (top right) top to bottom, (middle right) bottom to top, (bottom right) top and bottom to centre.

Once these points are found they are put into the existing circle fitting function that returns a circle object in the form of an x,y coordinate a radius and a standard deviation for the distance of the points from the circle. If the diameter of the circle is significantly lower than the width of the blob then it is assumed that the circle fit is not correct. In the case that it is fitted correctly the new distance, bearing and elevations are calculated from the circle that is returned. These can be significantly different to the original values calculated from the blob particularly when the image of the ball is heavily obstructed.



Figure 26: Circle fit to the ball can be seen in blue

### 3.3.5 Image Skew Correction

When the robot pans with speed or the ball is moving sideways, the shape of the ball in the image is distorted as can be seen in Figure 28. This distortion occurs because of the way in which the camera works. The pixels are read horizontally from the top left corner to the bottom right hand corner while the pixels are constantly being re-exposed. Therefore in the time between the exposing of the top line of pixels and the exposing of the bottom line of pixels, the balls place in the image may have moved dramatically. This causes problems with the circle fitting algorithm as the ball is no longer a circle, but ellipse.

To correct the skewing of the ball the idea used is to reverse this skew by skewing

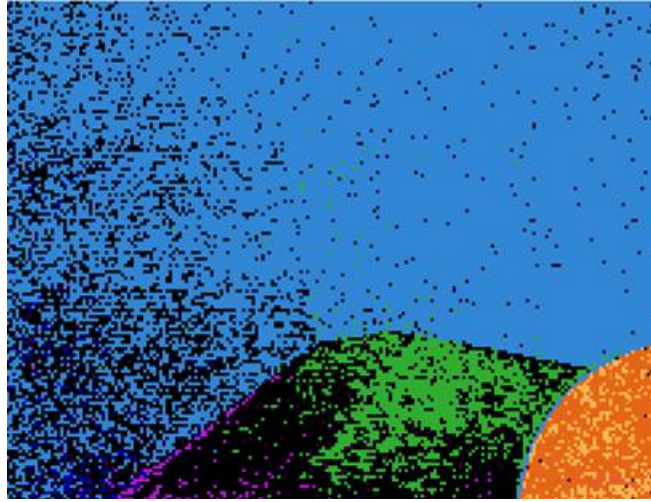


Figure 27: Circle fit of occluded ball shown in blue



Figure 28: Ball skew caused by head pan speed on stationary ball.

the image in the opposite direction by an equal amount thus nullifying the skew effect. To find the amount of skew that is present in the image the values available from the pan sensors we logged and evaluated these being the pan angle and the PWM signal sent to the motor. It was found that neither of these provided enough information to predict the skewing of the image. To predict the amount of skew it was found that the velocity calculated from the pan angle values provides the best indicator. Using the velocity the amount of skew can be computed. To do this first the effective camera distance ( $d$ ) is calculated using the image width ( $W$ ) and the

cameras field of view ( $FOV$ ) using the relationship shown in Figure 29. This value is also used to calculate the bearings of objects. The values given for the image width is 208 pixels and the field of view  $56.9^\circ$  :

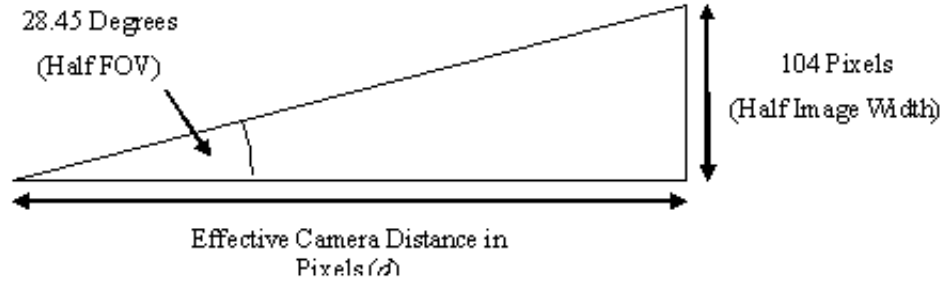


Figure 29: Calculation of effective camera distance ( $d$ ).

$$d = \frac{1/2 \times W}{\tan(1/2 \times FOV)} = \frac{104}{\tan(28.45)} = 191.9 \text{ pixels}$$

The effective camera distance can then be used to convert between an angle,  $\theta$ , and a pixel,  $x$ , as shown in Figure 30:

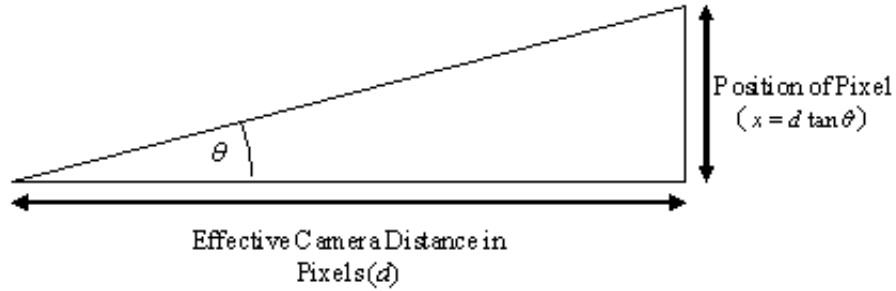


Figure 30: Pixel Calculation using angle.

To calculate the movement in pixels, firstly the position of a relative pixel is calculated from the pan of the frame before ( $\theta_p$ ):

$$x_p = d \tan(\theta_p)$$



The position of the current relative pixel is then calculated from the current pan ( $\theta_c$ ):

$$x_c = d \tan(\theta_c)$$

The position of the current relative pixel is then calculated from the current pan:

$$\Delta x = x_c - x_p$$

The next step was to find the time it took for the frame to be taken so that the distance that the pixel had moved during the capture of the image could be found. Using images of a fluorescent light which flickers at a known frequency (100Hz) the number of light dark cycles appearing on the image can be used to find the amount of time that has passed during the capturing of this image as seen in Figure 31. Using the fluorescent lights period of flicker ( $T_{FL}$ ) equal to 10ms and the number of cycles per image an estimate for the period of the camera ( $T_{cam}$ ) can be made.

$$T_{cam} \approx 3 \times T_{FL} \approx 30ms \approx 33.33ms$$

This suggests that the since the camera operates at 30 frames per second, or a period of 33.33ms, that the full time between images is used to read the next image. Therefore the time taken between scanning the top right pixel and the bottom left pixel is one full frame.

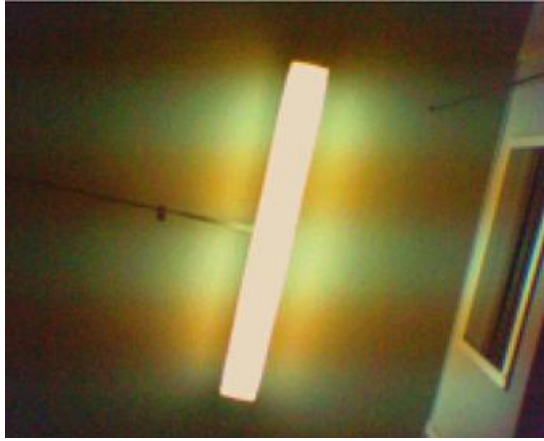


Figure 31: Image of the fluorescent light shows three distinct cycles.

Therefore the  $\Delta x$  value is used as is to calculate the skew amount per line.

$$\text{Skew per line} = \frac{\Delta x}{\text{Number of lines}}$$

These equations were implemented, and worked reasonably well correcting the skewing of the image of a majority of the images correctly. The major problem found was that other factors would affect the skew on the ball such as movement of the ball. Using this de-skewing technique the correction generally improves the distances returned by the circle fitting when the ball is stationary. One such result can be seen below in Figure 32.

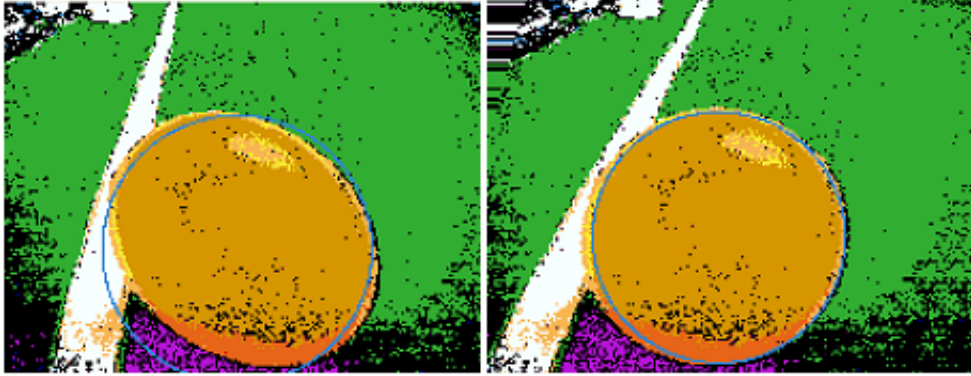


Figure 32: Classified image of ball with resulting circle fit before and after skew correction.

This method corrects the skew in the horizontal direction of the frame, however is much more difficult to implement in the vertical direction. This is because the vertical skew is caused mostly by the dropping of the robots body rather than movements of the head and involves the movement of many more joints which can mean increased noise and is also affected by external influences, eg if the robot is kicking the ball it will move at a different vertical speed compared to if there is no ball under it. The velocity of the ball also causes skew in the image that is not corrected using this method. This feature could possibly be used in conjunction with ellipse fitting to determine the velocity of the ball minus the velocity of the robots head.

### 3.4 Goal Detection

Since the introduction of soft colours and due to obstructions on the field, the goal is often not seen as a single blob, but a cluster of blobs. Because of this the blobs must go through a reconstruction process. Once the goal has been reconstructed a series of tests are performed and if the goal has been identified then goalposts can be found. Goal recognition also uses the confidence system, that is as the goal candidate performs well in tests it will increase in confidence and when it performs poorly it will decrease. At the end of the tests if the confidence is greater than zero the candidate is accepted as a goal.

#### 3.4.1 Goal Candidate Construction

The soft coloured blobs are linked to basic coloured blobs in a structure called a goal object. These blobs are all combined to get the total size of the object and stored as a goal candidate. These objects are then compared and those that are within a specified range, or those that line up horizontally are merged together to create one object as can be seen in Figure 33.

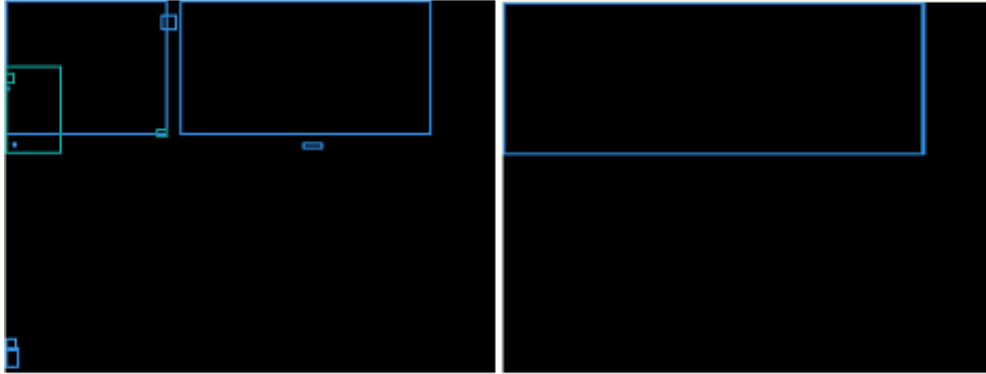


Figure 33: Left: Original blue blobs. Right: Resulting blue goal blob.

One problem that occurs is that when there is a goalkeeper inside the goals it can cause the blob to break into two with a gap at the location of the goalkeeper. To reconstruct this part the goal objects are rotated so that they are parallel to the ground to nullify tilts caused by the cameras orientation. Any blobs whose top and bottom co-ordinates are approximately equal are merged into a single goal

candidate.

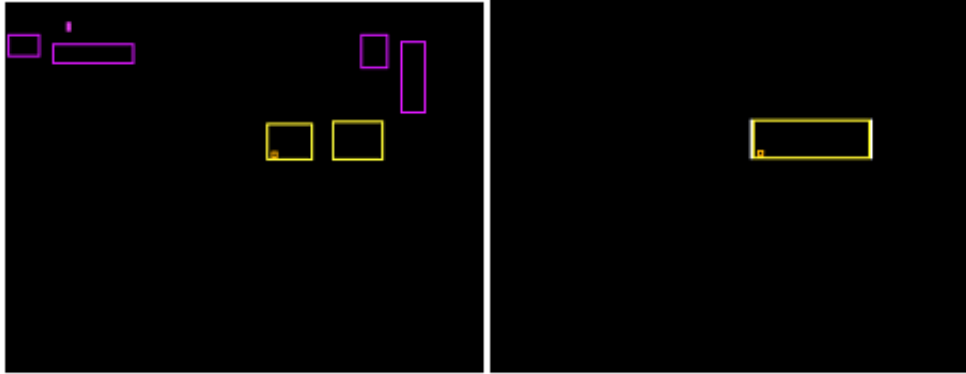


Figure 34: Left: Blobs of goal broken into two by goalkeeper. Right: Resulting goal blob.

### 3.4.2 Goal Candidate Cut-Off Detection

The sides of the goals that are cut-off can give a lot of information that is used in deciding what checks should be performed on the goals as well as in finding the goal posts. To find the edges that have been cut-off by the edge of the screen the blob is first rotated to remove the effects of the camera orientation. This allows 8 points to be selected. These points are the top, left, right, bottom, top left, top right, bottom left and bottom right these can be seen in Figure 35. These points are then un-rotated back into the original image to determine if they are close to, or outside of the edges of the screen. A side is seen as cut-off if two or more points on a side are located near the edge, or outside of it. For example if the top point and the top left point is outside or close to the edges of the screen then the top is assumed to be cut-off. This information on the suspected cut-off areas of the objects is used to tell the accuracy of the measurements obtained as well as the presence of goalposts in the image of the goal.

### 3.4.3 Ball Distance, Bearing and Elevation Calculations

The raw distance of the goal is found by using the height in pixels. This value is the only factor that will remain relatively constant when viewed from different parts of the field. The problem with this, however is that the robot must see the entire

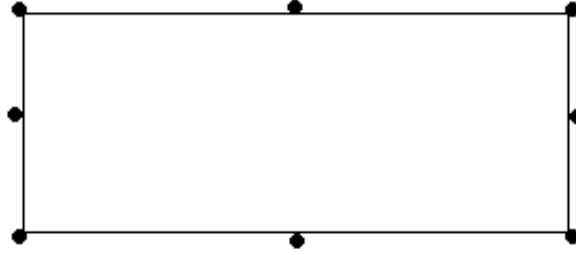


Figure 35: Points of blob used for object cut-off detection.

goal. When the robot is close to the goals this can be a problem. Because of this the infra-red distance sensors are used to gain a more accurate distance to the goal when the goal is close. The raw bearing and elevation is found using the centre of the goal blob or in the case of the goal posts their centre points. As with the ball these raw distances are translated back so they are relative to the centre of the robot. If the object cannot be properly seen and the cut-off detection has determined that part of the object has been cut-off that is required for correct measurements to be made, then the software sets a flag so that the localisation software knows to give these values less weighting.

The distance of the goal is calculated using the height of the blob formed around the goal. To do this an approximate linear relationship was used requiring an offset and a multiplier. This value was recorded for a large range of distances and these were plotted against the actual distance using Excel. Excel was then used to apply the line of best fit to these two sets of data and these equations were used to approximate the linear relationship between the two. Because the height of the goal is used, the distance of the goal becomes inaccurate when either the top or bottom of the goal is cut-off by the edge of the image. In this case a flag is set for the localisation component signalling that the distance given should be taken as a maximum limit on the distance, rather than the exact distance.

$$distance = \frac{multiplier}{goal\ blob\ height} + offset$$

When using this method it was found that as the robot moved closer to the goals distance does not remain linear. This occurs because as the robot moves towards the goals the sides of the goals artificially increase the height of the goal. It was

attempted to approximate the non-linearity by create two linear relationships, one near and one far, however it became difficult to determine when to use each one and it was found that there was a better way. Due to the introduction of soft colours, it was found that in many cases the sides of the goals were classified as soft colours. Therefore the height is taken only from the yellow or blue blob if the difference in heights is not too great.

#### 3.4.4 Goal Identification

Once the goals candidates are reconstructed they are then tested for identification. The following tests are only applied to goals that do not cover a large majority of the screen since it can be very difficult to get readings in this situation. For this reason if a goal candidate appears to be a very close it is assumed to be a goal and these tests are skipped since it is very likely to be the goal as there are not usually other objects large enough and of that colour to fill the cameras FOV. The first test that is performed is a colour test similar to that used in the ball recognition (Section 3.3.3). This test determines that the candidate is on the field.

Because the goals are stationary, like the ball, their height should remain relatively constant. For this reason the z-value is calculated (Section 3.3.2). This is then compared to a range of acceptable values. If it is not in this range then the candidate is penalised through a reduced confidence. Elevation checks are then performed; this is because the range of z-values for an acceptable goal needs to be quite large to account for incorrect distances and elevations. Therefore the elevation of the objects is used to help remove the other false goals.

#### 3.4.5 Goal Post Detection

Using the information gathered during the cut-off detection the image is scanned for goal posts. This is done by performing a least squares line fit on the edge of the goal. Firstly the points to be used in the fit are gathered. This is done by performing scans parallel to the horizon at regular intervals along the side of the goal. The least squares line fitting method is then performed by first calculating the sum of squares as this method appeared the simplest to implement as an algorithm. The values used with the sum of squares method are:

$$ss_{xx} = \left( \sum_{i=1}^n x^2 \right) - n\bar{x}^2$$

$$ss_{xy} = \left( \sum_{i=1}^n x_i y_i \right) - n\bar{x}\bar{y}$$

$$ss_{yy} = \left( \sum_{i=1}^n y^2 \right) - n\bar{y}^2$$

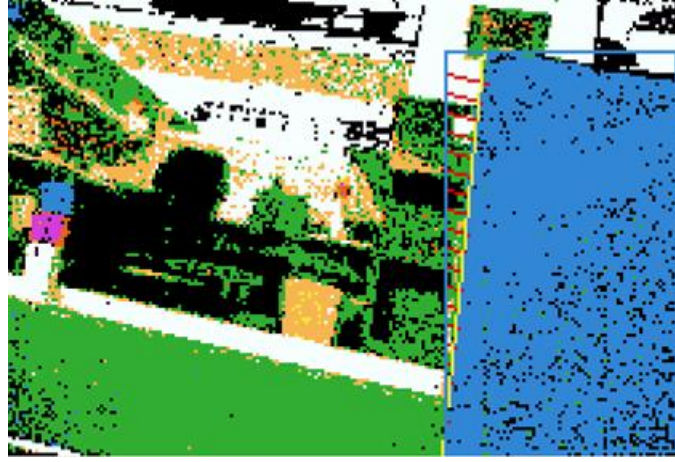


Figure 36: The scanning path for the line fit points are drawn in red, while the least squares line fit is drawn in yellow.

These are used to calculate the equation of the line  $y = a + bx$ . Firstly the regression coefficient  $b$ , the correlation coefficient  $r^2$ , and the residual variance  $s^2$  are calculated as follows:

$$b = \frac{ss_{xy}}{ss_{xx}}, \quad r^2 = \frac{ss_{xy}^2}{ss_{xx}ss_{yy}}, \quad s^2 = \frac{ss_{yy} - \frac{ss_{xy}^2}{ss_{xx}}}{n - 2}$$

The value of  $a$  is then found:

$$a = \bar{y} - b\bar{x}$$

where :

$$\begin{aligned}\bar{x} &= \sum_{i=1}^n x_i \\ \bar{y} &= \sum_{i=1}^n y_i \\ \overline{xy} &= \sum_{i=1}^n x_i y_i\end{aligned}$$

Once this line has been fit, the correlation co-efficient,  $r^2$  and the sample variance  $s^2$  are used to determine the edges suitability as a goal post. One change that had to be considered was that in the most likely and most ideal case of a goal post being a straight vertical line, the gradient of the fit line would be infinite. To correct this problem the x and y axes are swapped in the line fitting mathematics and a check is done in the case of a completely horizontal line to prevent a divide by zero error that would cause the robot to crash. The raw distance for the goal is used and new raw elevations and bearings are calculated for the individual posts using their centre points. The detected posts can be seen in Figure 33 and Figure 34 they are shown as the vertical white lines on the edges of the goals.

### 3.4.6 Goal “Gap” Detection

The locations of the goals play a large part in the NUbots kicking behaviour in that before a robot takes a shot at goal it will first try to visually line up the goals. However the centre of the goal (given by the goal detection) is not always the best place to kick the ball. The best place to kick the ball in a majority of cases is at the largest “gap”, the goals largest unobscured area. This allows the shot to go past the obstacles and therefore increase the chance of scoring. To help improve the NUbots goal kicking abilities when faced with a goal keeper or other obstacles, goal gap detection was developed.

The gap detection consists of an algorithm that searches from the left hand side of the goal to the right hand side of the goal in the image designated by the current goal blob. The search runs parallel to the horizon and hence roughly parallel to the goals. The height of the scan is calculated depending on the height of the goals in an effort to scan at approximately the shoulder height of another robot, thereby at the height of the largest portion of the robot. Runs of pixels that are of the correct



goal colours are counted and the longest run found is given as the gap. This gap is given as a field object and therefore the robot can use this to line up shots.



Figure 37: Scan path for the gap detection is show in yellow.

### 3.5 Blob Rotation

The rotation algorithm modifies blobs to simulate the camera of the ERS-7 robot being horizontal. This conversion simplifies the object recognition code

The rotation calculations are as follows:

$$\begin{aligned}
 horizonLine.Angle &= \tan^{-1} horizonLine.gradient \\
 offsetX &= \frac{IMAGEHEIGHT}{2} \\
 offsetY &= horizonLine.gradient \times offsetX + horizonLine.offset \\
 X &= X - offsetX \\
 Y &= Y - offsetY \\
 rotatedX &= X \times \cos(-horizonLine.Angle) - Y \times \sin(-horizonLine.Angle) \\
 rotatedY &= X \times \sin(-horizonLine.Angle) + Y \times \cos(-horizonLine.Angle) \\
 rotatedX &= rotatedX + offsetX \\
 rotatedY &= rotatedY + offsetY
 \end{aligned}$$

The offsets are used to virtually move the blob into the centre of rotation. OffsetX is the centre X of the image, and OffsetY is where the X offset intersects the horizon line.

When rotating blobs it is important not to simply take the corner points in the calculations. Doing so cuts out useful information and introduces garbage inside the rotated blob. To understand this, Figure 38 shows the information stored about a blob when it is formed from the classified image.

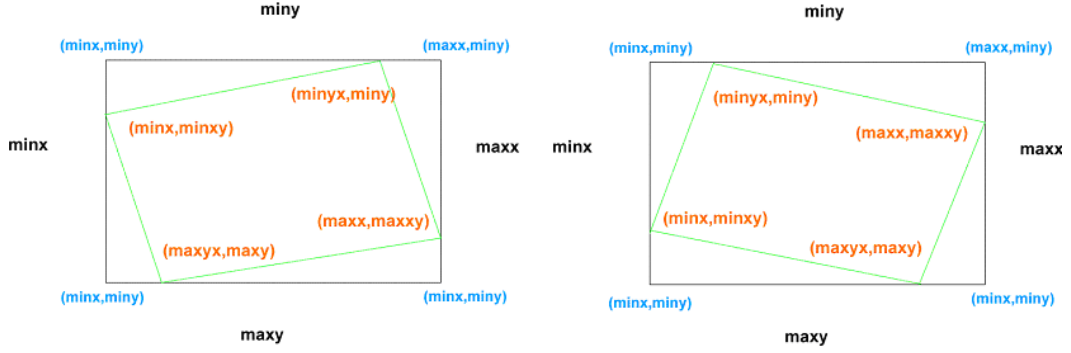


Figure 38: Orange text are the real co-ordinates, Blue text are the blob corners.

From the above illustration we would rotate the real coordinates of the blob (orange text) if the real coordinates are not the corners. By doing this the rotated blob would have maximum coverage of the real colours, equivalent to if the classified pixels were rotated before blob creation. In cases where the angle of rotation is large, the code ensures that the rotated coordinates correctly derive the variables of minx, maxx, minyx, maxyx, minyx, maxy, minxy and maxxy. As transform position relies on the inverse-rotated coordinates, a function was created to allow the raw coordinates to be calculated.

The inverse-rotation calculations are as follows:

$$horizonLine.Angle = \tan^{-1} horizonLine.gradient$$

$$offsetX = \frac{IMAGEHEIGHT}{2}$$

$$offsetY = horizonLine.gradient \times offsetX + horizonLine.offset$$

$$rotatedX = rotatedX - offsetX$$

$$rotatedY = rotatedY - offsetY$$

$$X = X \times \cos(horizonLine.Angle) - Y \times \sin(horizonLine.Angle)$$

$$Y = X \times \sin(horizonLine.Angle) + Y \times \cos(horizonLine.Angle)$$

$$X = X + offsetX$$

$$Y = Y + offsetY$$

### 3.6 Filtering Algorithm

The filtering algorithm aims to remove noisy blobs from vision. The algorithm was written in robot recognition to reduce strain on assigning robot blobs to robot containers. Eventually it was separated so that it could be used for filtering beacon blobs.

The filtering algorithm is called after the rotation algorithm. It examines each blob and decides whether it should be ignored or merged with another blob by using threshold values. Figure 39 illustrates the scenarios catered for in the filtering algorithm.

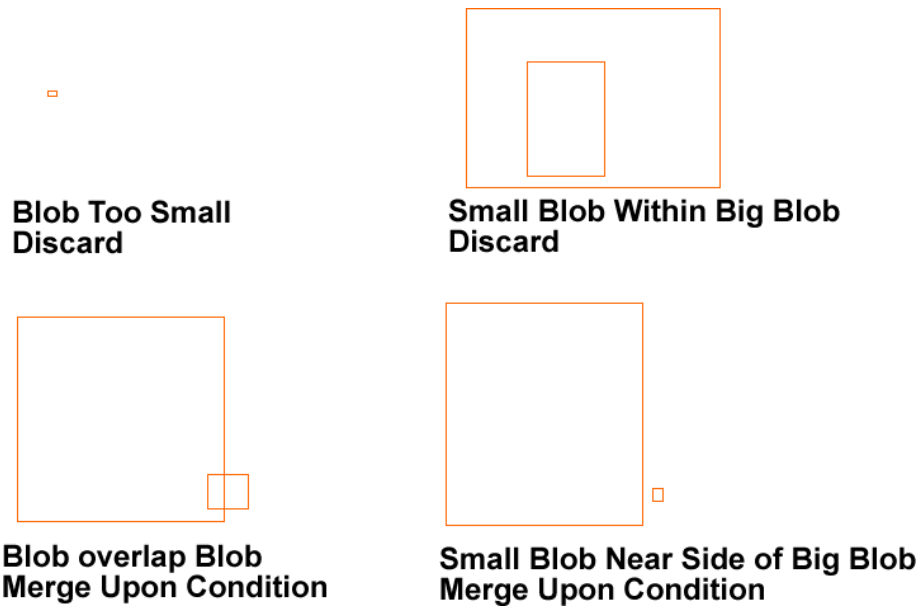


Figure 39: Outlines the purpose of the filtering algorithm. Merge upon condition require rules such as one blob should be sufficiently larger than the other for them to become one; otherwise you would end up with a very large blob where object recognition has no useful information to process.

### 3.7 Beacon Detection

The beacon detection code was originally developed within Python. The code searched for individual beacons by checking all pink and yellow blobs for pinkYellow (Pink = Top, Yellow = Bottom) and yellowPink (Yellow = Top, Pink = Bottom) beacons, then all the pink and blue blobs for pinkBlue and bluePink beacons. These searches operated in roughly a  $O(4n^2)$  order operation, with  $n$  being the number of blobs in the image. The additional computational load of having all the beacons visible resulted in a reduction of the frame rate from the standard 30 fps to around 18 fps. This drop in frame rate results in the dog responding slowly to its environment.

The next evolution of beacon recognition searched for beacons by iterating through the pink blobs. Each pink blob is then compared to the blue and yellow blobs found in the image. This reduces the computation time from  $O(4n^2)$  to  $O(2n^2)$ . By incorporating the goal recognition code into the current beacon code, the detection of goals was increased. This was due to the fact when a blue or yellow blob is not accepted as a possible beacon, it is then it is a candidate to be a goal. By scanning for white below the pink blob, allowed the code to determine whether the pink blob was a top or bottom beacon blob. The choice of white was because colour classification of white is simple, and the colour beneath the bottom beacon blob is white. An additional speed improvement was made when it was decided that all beacon blobs would be sorted from largest to smallest.

Even though the changes made to the python code removed the frame drop problem, it was decided to keep the combined beacon and goal code in C++. The beacons and goal code was later split again to retain logical design structure and allow group members to work on their individual sections.

After the 2005 Australian Robocup Open at Griffith University, a number of problems was discovered with the beacon code. It was noticed at the Australian Open that the previous evolution of beacon recognition missed beacons. This was due to the scanning for white on the classified image beneath a pink beacon blob. At Griffith University the green soccer field is much lighter in colour, resulting in the classification of green on areas which were white. Since the check for white determines that the pink blob is on the bottom, beacons like yellowPink and bluePink were difficult to recognise.

Before reaching the current evolution, a test bench was made in scanning for

yellow and blue, above and below a pink beacon blob from the classified image. Out of 10 beacon frames, in good cases 6 were recognised, in average cases 4 was recognised. This is due to colour classification either misclassifying the colours or under classifying the images.

The current evolution takes concepts found in the initial code with those of previous evolutions. The initial code didn't require scanning of classified image, which is incorporated in the current evolution. Previous evolutions of the detection code had  $O(2n^2)$  operation times. The current evolution only operates at this level in worst case scenarios.

The current evolution iterates through pink blobs, taking the largest first. It first checks the pink blobs against yellow blobs then blue blobs. To improve running time the yellow and blue blobs are first sorted size, largest to smallest. The recognition code determines which blobs are top and which are bottom beacon blobs by comparing their relative positions. When two blobs are recognised as a beacon, they are marked as used, and hence ignored by all other tests. When both types of beacon are found for one colour type (eg yellowPink and pink pinkYellow) the code stops searching for anymore possible beacons of that colour combination.

### 3.7.1 Beacon Distance, Bearing and Elevation Calculations

Once two blobs satisfy the checks to be regarded as a beacon, the two blobs are used to create the beacon field object. Parameters such as centreX, centreY of the beacon field object are determined by using the information from both blobs; for instance, the centreX, centreY of the beacon field object is calculated using the two blobs.

The beacon field object is then passed into transform position to determine its elevation, bearing, and distance. Transform position consists of matrix calculations that involves translations and rotations, and it requires tuning a calibration constant. The calibration constant is a mathematically derived ratio that requires tuning due to inconsistencies in the camera distance, colour classification errors, and slack in the sensor inputs. For beacons, the calibration constant is a number divided by the distance between the centre of the beacon blobs.

Since elevation relies on centreY of the beacon field object, bearing relies on centreX of the beacon field object and distance relies on distance between the two

blob centres, their calculations are consistent if ideal beacon blobs are used.

### 3.7.2 Non-ideal beacon blobs

Non-ideal beacon blobs can distort distance, bearing and elevation values which affects localisation and therefore behaviour. This is illustrated in Figure 40.

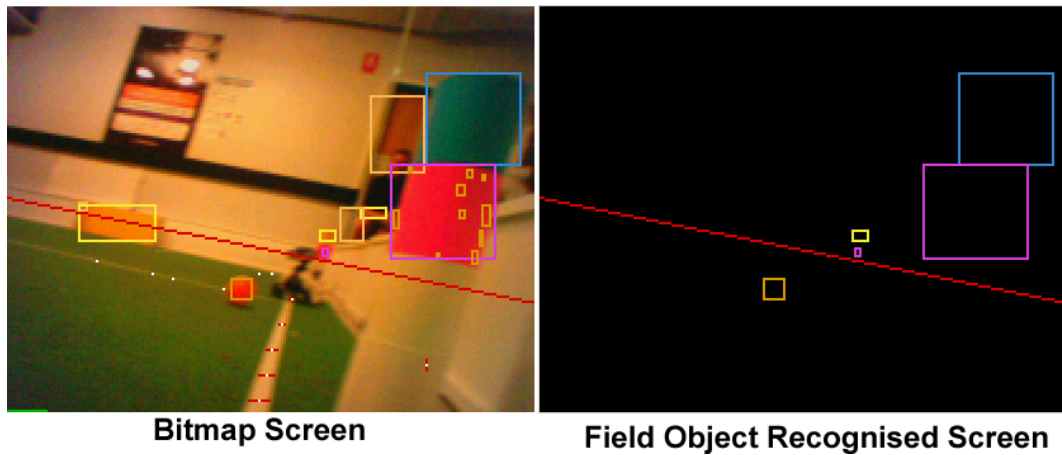


Figure 40: The further away the beacon blobs are, the more non-ideal it becomes. The images here are continued in the next figure.

Non-ideal beacon blobs stems from colour classification, and is a hard case to tune, as mapping more raw colours may allow more incorrect blobs to be formed elsewhere in the image. The solution developed was to make the beacon recognition code able to take bad beacon blobs and turning them into ideal beacon blobs. Take the instance of a small beacon blob matched with a big beacon blob, formed by colour classification. If these two blobs pass sanity checks to become a beacon field object, then project the small blob to the size of the big blob. This will improve distance, bearing and elevation calculation and provide consistency in the recognition. This is illustrated in Figure 41.

## 3.8 Robot Detection

The previous attempt on robot recognition did not provided good results, and was very limitedly used in the 2004 Robocup competition.

This year, we have added orientation recognition as a new feature in the robot

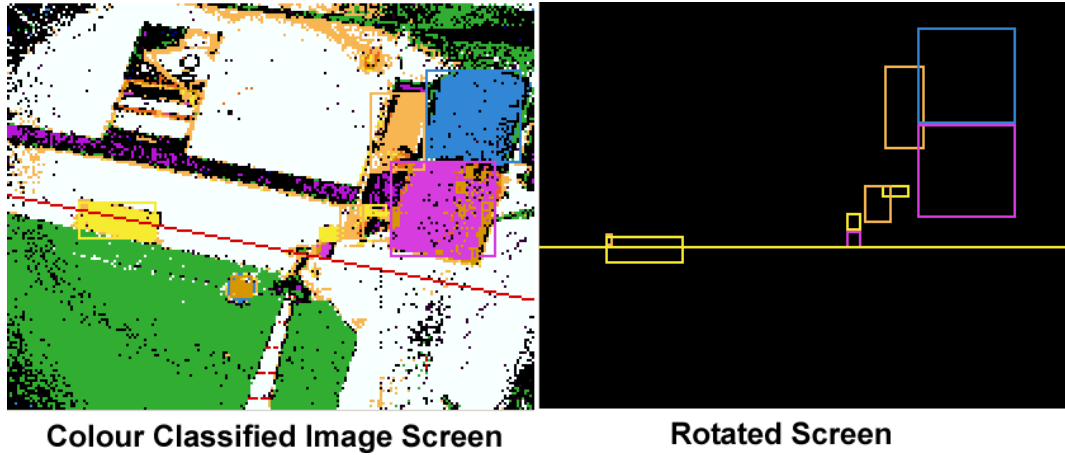


Figure 41: Small beacon blobs projected to be big beacon blobs. This creates an ideal scenario, and allows distance, bearing, elevations to be consistent and accurate. The projection is performed after rotation. The screen on the right is the rotated screen. The bitmap image is in the previous figure.

recognition code. Orientation recognition relies on having sufficient information of the blobs from the classified image to estimate the directional stance of the robots on the field, in particular opponent robots. By knowing the opponent dogs' directional stance, we can implement better game play behaviours such as passes behind opponent robots.

### 3.8.1 Robot Clusters and Containers

The general method of robot recognition begins by finding clusters of robot blobs on the field. Each identified cluster may represent an ERS-7 robot. Clusters of blobs like this are assigned assigned to a robot container. The robot container is a data structure that stores at most three robot blobs, which is sufficient for the code to estimate a robot's orientation. The sufficient condition was derived by observing the possible number of patterns formed by the uniforms as an ERS-7 robot is viewed from different perspectives.

The initial attempt of assigning robot blobs to robot containers worked on an arbitrary sorted order of robot blobs. This meant that the first robot blob may be located on the top left corner; the second robot blob may be located on the bottom

right hand corner; the third robot blob may be located on the bottom left corner; and so on. The reason for the arbitrary sorted order came from the blob formation code that forms and merges blobs by scanning top-down, left to right [6].

An approach to assigning robot blobs to robot containers was using sanity factors, which switches on when there is at least one robot blob in a robot container. Sanity factors are values that are similar in respect to a person's confidence. The confidence increases when certain factors are met and decrease when they are not. The factors are conveyed through threshold values in the robot recognition code. The sanity factors work by examining the current robot blob, then checking relationships, (such as ratios between the current robot blob area and distance to all the other robot blobs in all the robot containers) then assign the current robot blob to the appropriate robot container based on sanity factor values. This was tedious and time consuming as thresholds that work with sanity factors required constant tuning and testing.

An improvement in assigning robot blobs to robot containers was made by sorting the robot blobs from left to right. This lessens the tuning of threshold values as results are more favourable. The results are more favourable because robot containers are created sequentially. That is, the current robot blob would be compared with only the current robot container, rather than with all the robot containers. If the current robot blob appears to be another ERS-7 robot, a new robot container is created. Further improvements began when scanning of classified image was used. The robot code scanned three areas between each robot blob for colours of green or orange; if there are colours of green or orange, a new robot container is formed. This is illustrated in Figure 42.

### 3.8.2 Orientation Recognition

Orientation comes into play when the assigning code has completed running. We take each robot container, and examine the information regarding the stored robot blobs. Information such as largest to smallest in area and highest to lowest in height are stored and calculated within the assigning code, ie performed on-the-fly. These information aids with the robot recognition code to determine their orientation. The robot recognition code attempts to distinguish the orientations: Left, Right, Front, Side, Back and Unknown using methods similarly described in Section 3.8.1. In particular scanning of the classified images contributes to the sanity factors that



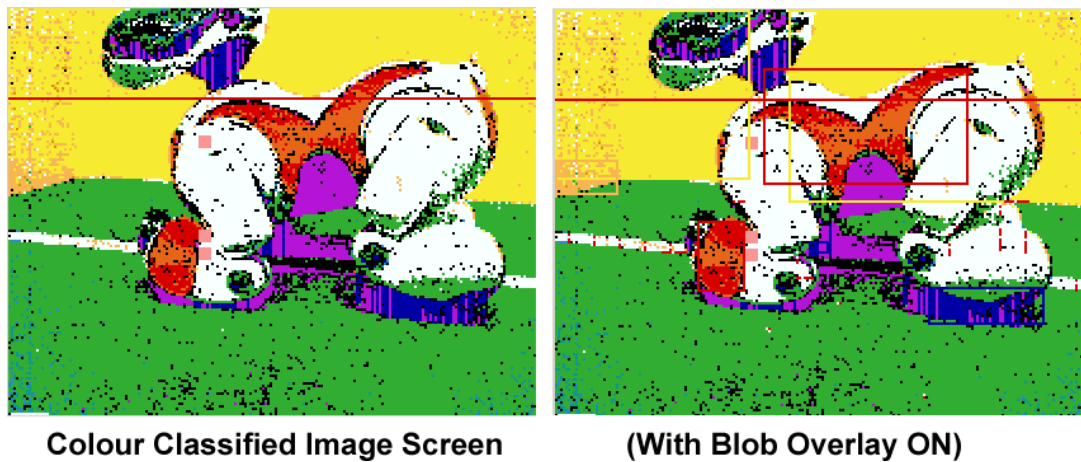


Figure 42: The scanned area is indicated by the solid pink squares. Since the colour green and orange are absent from the scanned area, the robot code would allocate the two red robot blobs to one robot container.

determine the robot's orientation.

Side orientation is when the ERS-7 robot has moved too close to the side of another ERS-7 robot that it lacks sufficient information to determine whether the other robot is facing left or right. One reason is because the front leg patch is outside the image. Unknown orientation is necessary as not every pattern of robot blobs can be used to determine its orientation.

### 3.8.3 Red Robots

The orientation of robots with red uniforms is easier to tackle as red is a distinct colour and thus red robot blobs are formed around patches of classified red pixels. This is illustrated in Figures 43-48.

### 3.8.4 Blue Robots

The colour of the blue uniforms is hard to distinguish from other dark objects and shadows on the field, therefore edge detection is used for detecting robot blue. The Sony ERS-7 AIBO robots are white, in RoboCup competition the blue team wears a navy blue uniform, the contrast of brightness between the two YUV colours is distinct and a definite edge results.



Figure 43: Orientation: Left.

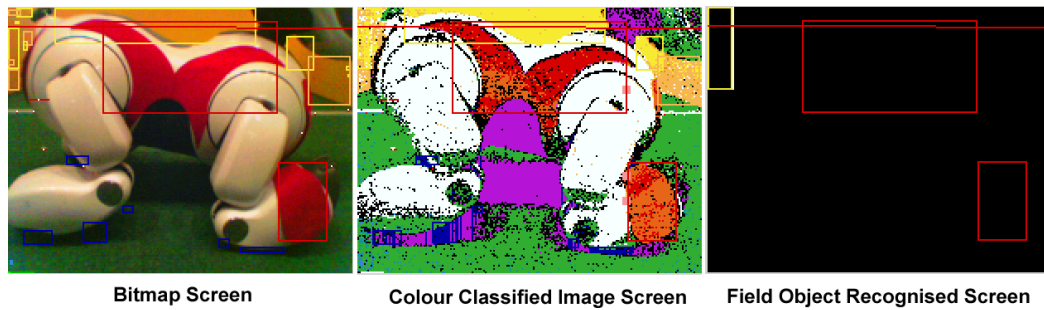


Figure 44: Orientation: Right.

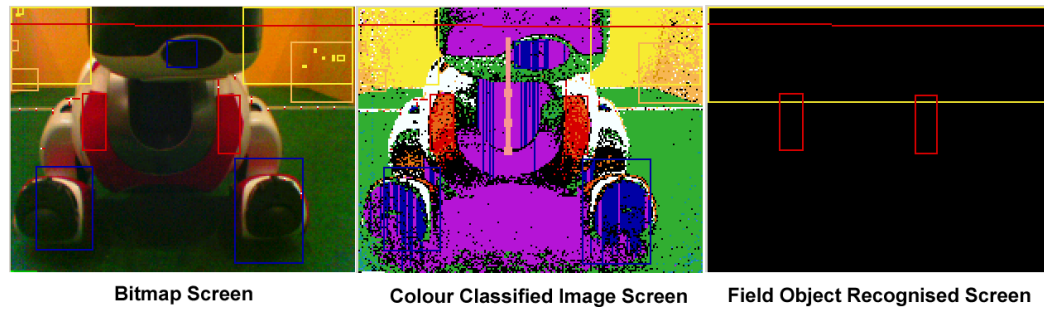


Figure 45: Orientation: Front.

The blue uniforms are initially classified as shadow. This is due to the fact that the blue uniform has YUV values close to that of the corners of image, dark corners of the blue goal, object shadows on the field and the dark shades of every robot. However, all of these objects do not have the same edge-colour characteristics.

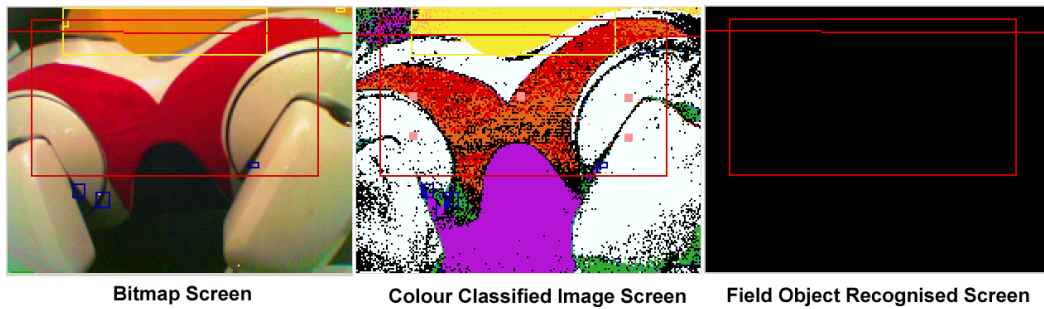


Figure 46: Orientation: Side.

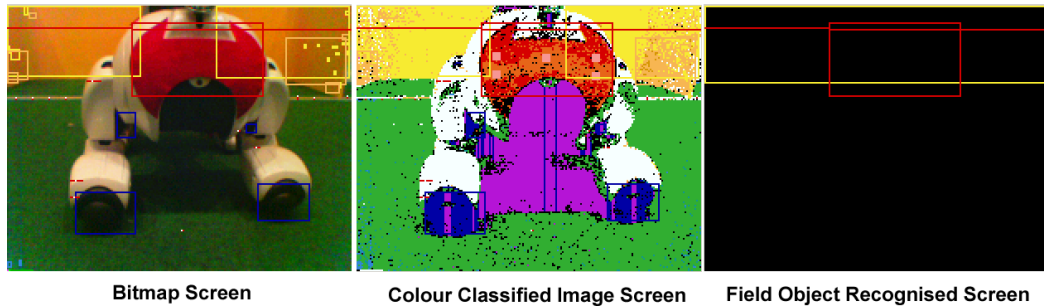


Figure 47: Orientation: Back.

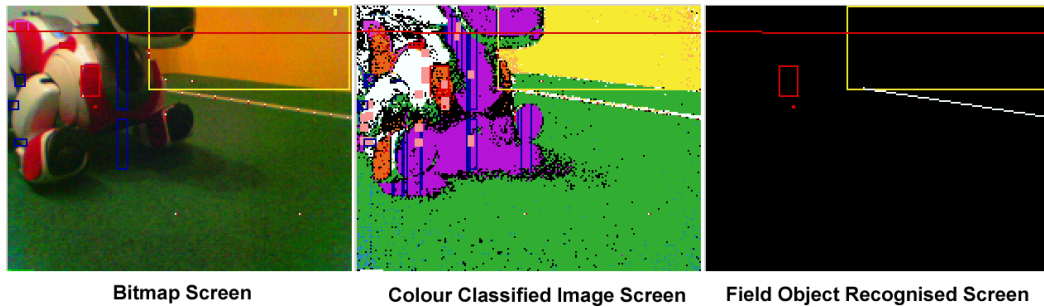


Figure 48: Orientation: Unknown.

After initial colour classification is performed, secondary classification is carried out to classify robot blue.

The general edge-colour characteristics were derived from studying images similar to that of Figure 49 YUV image, edge-classified image, colour classified image,

secondary classified image of blue robots.. There is always a definite edge between the white robots and the blue uniforms hence this is the correlation that is to be searched for.

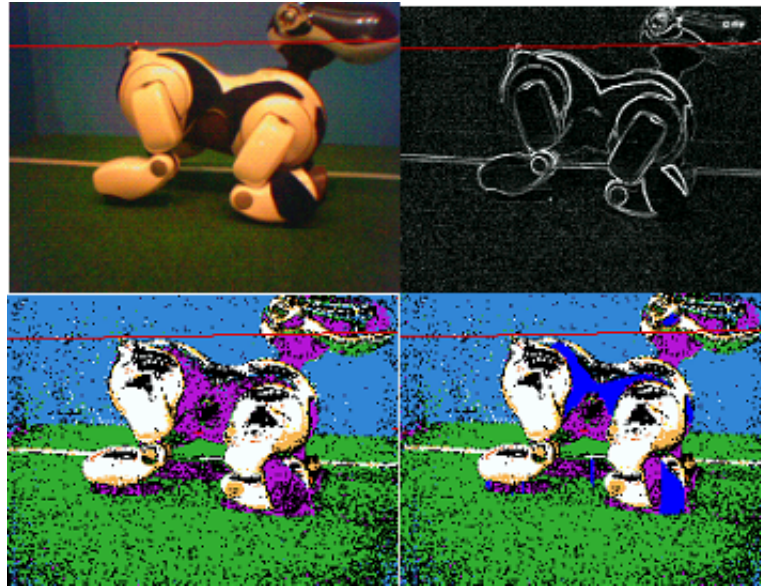


Figure 49: YUV image, edge-classified image, colour classified image, secondary classified image of blue robots

For the width of the image, the image is scanned downwards. If white is found a negative edge is searched for, if a negative edge is found the process then checks what colours follow white. If a run of shadow colour is found then the pixels are updated to be robot blue until a positive edge is found. However, if a run of another colour occurs then it will break the check. If a run of another colour is found after white is found it will not change pixels to robot blue. If the run is found after a run of shadow colour then it will break the pixel modification. Note: an edge between blue and green is not recognised as there is a slow transition of similar shades similar shades.

One of the most important aspects that the robot recognition code has to cater for is the situation that blue robot blobs being formed from shadow colours of red robots. Early versions of the robot recognition code did not cater for this, and robot blue was detected from the shadows of the robot red.



Figure 50: Orientation: Left.

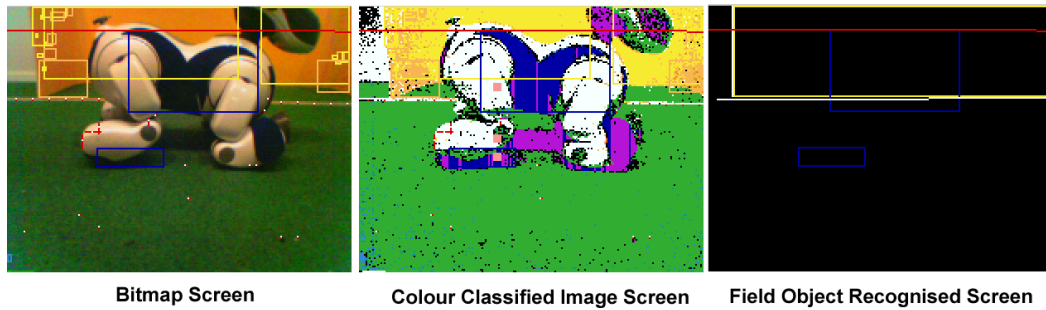


Figure 51: Orientation: Right.

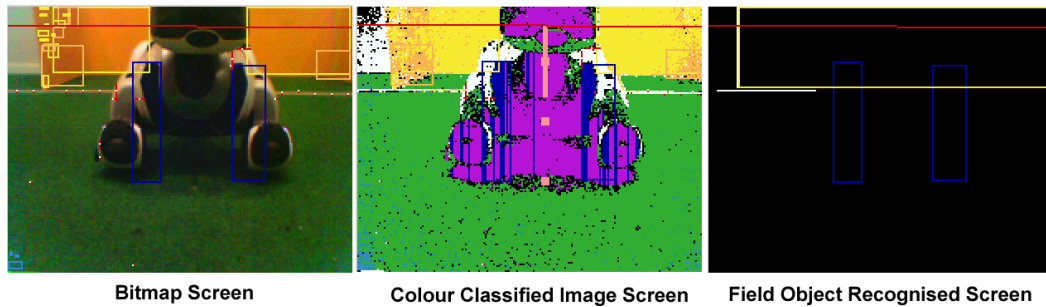


Figure 52: Orientation: Front.

### 3.9 Line Detection

The detection of field lines is approached in a completely different process to that of Blobs and Objects. When searching for objects, sections of a specific colour are joined together to form a blob. While this works well for most objects, white causes





Figure 53: Orientation: Side.

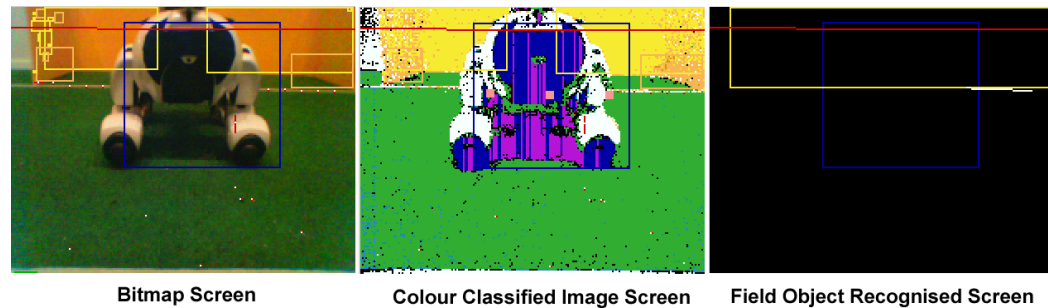


Figure 54: Orientation: Back.

problems due to its abundance in the image and hence is normally ignored. Also the thin and long nature of lines makes blob created difficult due to missed pixels in the classified image. And even if a line can be found as a blob, the blob is useless since it is likely to cover most of the screen.

To over come these issues the detection of lines is based around the unique features that field lines have. All field lines share two properties which make them easier to distinguish. Firstly, the lines all have green on either side of them. This allows a simple test when searching for the lines. Secondly, the lines all are long and thin. This means that every pixel doesn't need to be searched; instead a widely spaced search grid can find points on the lines, which can then be joined back together to form the lines. This reliance on points also allows the lines to be partially hidden behind another object without effecting their detection.

With these factors in mind, the image can now be efficiently searched in the following steps.

### 3.9.1 Point detection

The image is searched using a horizontal and vertical search grid with a distance of 10 pixels. The distance can be tuned to determine the ideal distance between points on the lines, however 10 pixels was discovered to be acceptable. Since all lines will be on the field, the search only needs to occur below the horizon line. By doing searches in both the horizontal and vertical we ensure that all the lines will be bisected at worst a 45 degree angle, giving a short transition which can easily be detected and marked. To increase the accuracy of this search, the initial pass looking for transitions is performed on the edge classified image. Due to the large colour change between the green of the carpet and the white of the lines, field lines appear very distinctly on the edge classified image. By searching this image first it was found that the points were more consistently found and the width of the lines more accurately averaged. This also allows for when the classified image isn't completely tuned to the venue since the transitions are not required to be exactly classified into field green or white. Once an edge is detected the classified image is checked to confirm the edge contains the necessary Green/White/Green transitions. With an edge found, the classified image is used to determine if it is a field line transition. To avoid classification problems the 5 pixels on either side of the line are averaged for the needed colours. This does result in a few points being needlessly rejected, but does result in the most robust detection method.

### 3.9.2 Line creation

Once the points on the image have been found, the lines need to be created. To make sure points not contained within a line are not added to a line, each point is checked. Checking each point against all others would be the best for creating the lines, however such an operation would be an  $n!$  operation. To avoid this, points are only checked against other points from the same search direction (eg from the vertical search, or the horizontal search) and against points no more than 30 pixels to the right/down. This means that a line can be formed even when there is a 30 pixel gap between each point. Once the two points are found which may form a line, the gradient to each new point is checked against the current gradient of the line. If the new point's gradient is within a tuneable margin of the created line, then the point is added to the line. The gradient has to be checked with a margin

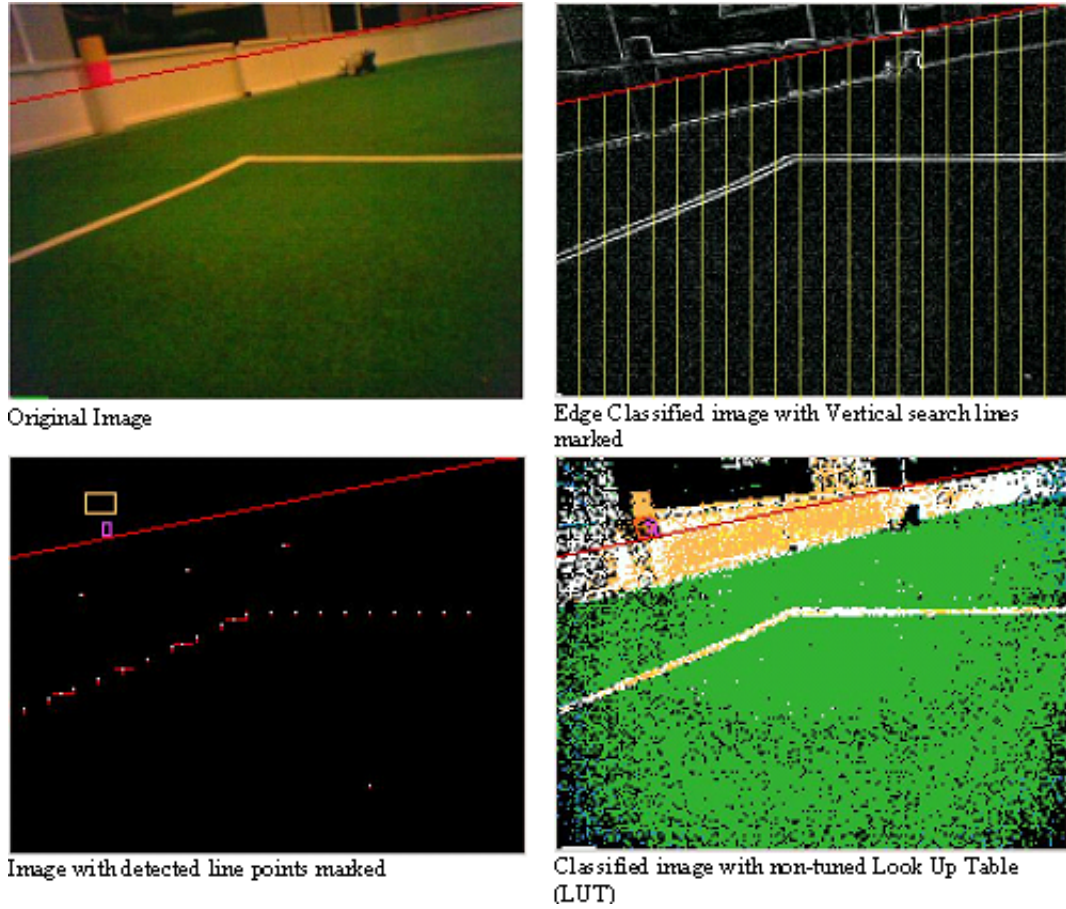


Figure 55: Line detection.

since pixilation and skewing can distort the point's position by a pixel or two. Once a point has been allocated to a line which contains more than five points, the line is marked as valid, and all the points contained in that line are marked as used. This means that points can not be part of more than one line. The lines then have their equations correctly worked out. Up till this point the line creation process has been avoiding the use of Trig functions due to their speed and processor restraints. However once valid lines are found the lines equations are needed to allow further calculations, and hence the line of best fit is calculated. While the lines found at this point are all valid lines, it often occurs that a single long line may have been split into two lines. This occurs when an object such as another robot is blocking part the view of the line. To overcome this problem the lines are each tested to



determine if they are actually two parts of the same line. Lines which are found to be are joined by moving all the points from one half to the other, and removing the point less line. The new line is then re-calculated with all the points.

### 3.9.3 Corner detection

While the finding of lines is important, the corners that lines form are much more useful. When trying to use field lines in localisation, the Kalman filter can use the data provided by a know point much more effectively then that of a line alone. To allow this the interception point of each found line is calculated to determine corners. The lines often need to be extended to form a correct corner, which requires checking to make sure a line such as that of the penalty box is not being extended to the sideline to create a virtual corner. The discovery of a field line corner is useful, it still needs to be uniquely identified to be helpful to localisation. When a corner is located the details of that corner are not known. The corner point is first tested to determine if it is an L corner or a T corner. Because of the low number of T corners on the field, T corners are much easier to identify, since a goal will normally also be in the image. L corners on the other hand need much more information on them to discover which corner is being seen. To identify the corner it is first calculated to be an inside (eg the dog is in the acute angle formed by the line) or and outside corner. This is done by working out the start and end points of the lines forming the corner and testing them against the position of the screen. If the both end of the lines are above the corner, the dog is outside, if both are below the dog is inside. This test works well for most instances, however when one point is above and one below test need to be make as to the end points left or right position compared to the corner point. Once the L corner is identified as an inside or outside corner, other visual clues are look for. In the case of the penalty box, most outside corners also give a view of a goal, and another T corner (where the penalty box connects to the side line.) and hence provide two excellent points to allow the dogs to localise. Looking out from the goal the corner is identified as being an inside corner, which allows the dog to partially localise based on seeing one of 4 known corners. If a beacon is also contained in the image, the corner can be uniquely identified which allows the dog to localise much more precisely.

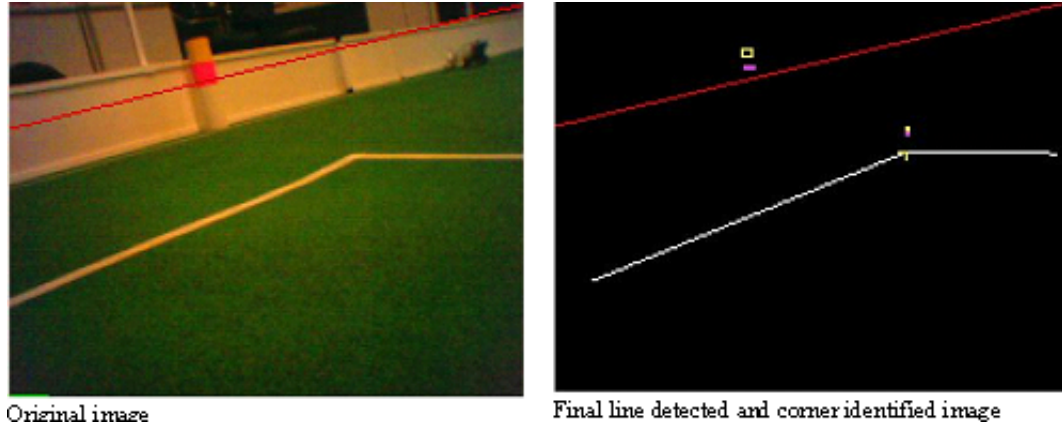


Figure 56: Corner point detection.

### 3.10 Identifying potential obstacles using scan lines

Scan lines involve searching around the horizon to determine what lies in image. We do credit the basic idea to the 2004 German Team [Röfer *et al.*, 2004], but our approach was developed without looking at their code or reading the section of their report.

Essentially lines are drawn up and down from the horizon, the colour of each pixel on the line is given a score, the sum of which determines a ‘likelihood of obstruction’ on that line. For example the colour field green indicates no obstruction (score=1.5) as does a goal colour (score=1), robot uniform colours indicate an obstruction (score=-15) while white is also considered an obstruction (score=-1).

A line with a total score of less than 70% of its height is considered obstructed. We group the ten lines into five sections: *far left*, *left*, *middle*, *right* and *far right*.

Behaviour can then query which parts of the image are obstructed, as an example the dribbling code checks *middle* to see if the area in front of the robot is clear.

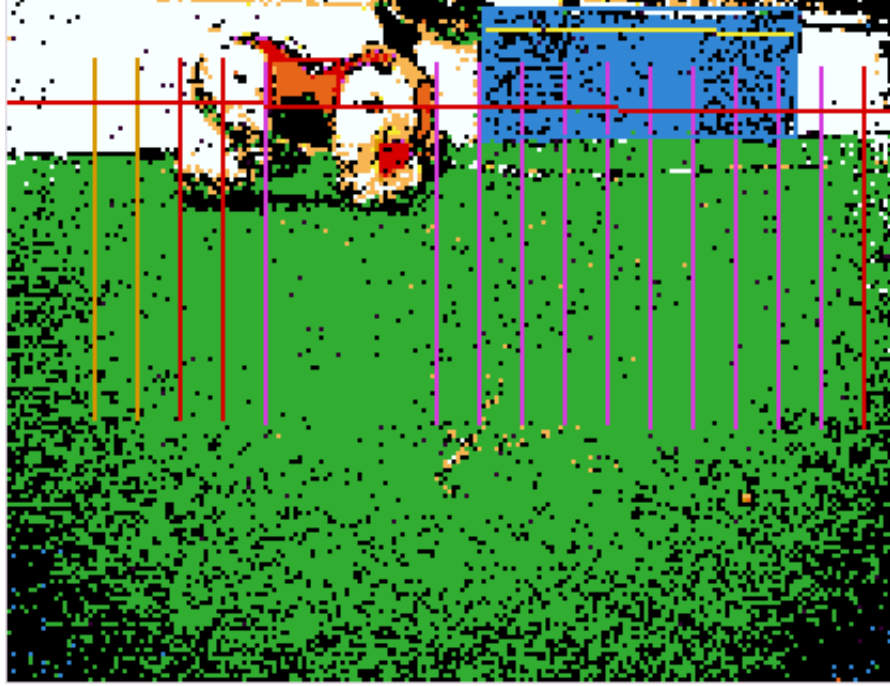


Figure 57: Scan lines on an image. Pink lines (>85%) indicate no obstruction, orange lines (70-85%) have some obstruction, red lines (55-70%) an obstruction and no line (<55%) indicates heavy obstruction.

## 4 Localisation and World Modeling

The NUBot localisation system is based on an Extended Kalman Filter (EKF) to provide estimates of the robot's position and the ball's location. Variance estimates are retained by the EKF and used in the calculation of confidence of distances and bearings to other objects,

### 4.1 Kalman Filter

The EKF is a seven-state filter based on the traditional EKF equations.  $\hat{x}_{k|k-1}$  is an estimate of the robot's position at frame  $k$  given the information up to frame  $k-1$ , and  $P_{k|k-1}$  is the error covariance matrix of this estimate.

Time Update:

$$\hat{x}_{k|k-1} = f(\hat{x}_{k-1|k-1})$$

$$P_{k|k-1} = A_k P_{k-1|k-1} A_k^T + Q_k$$

where  $f(x)$  is a function that updates the estimate  $x$  of the robot's position last frame to the new position based on the odometry received from the locomotion engine,  $A$  is the Jacobian of this function, and  $Q$  is an estimate of the model covariance (that is, the possible error in the model combined with our belief of the variance in the odometry).

Measurement update:

$$\begin{aligned}\bar{y}_{i,k} &= h_i(x_{k|k-1}) \\ J_{i,k} &= P_{k|k-1} C_{i,k}^T (C_{i,k} P_{k|k-1} C_{i,k}^T + R)^{-1} \\ \hat{x}_{k|k} &= \hat{x}_{k|k-1} + J_{i,k} (y_{i,k} - \bar{y}_{i,k}) \\ P_{k|k} &= (I - J_{i,k} C_{i,k}) P_{k|k-1}\end{aligned}$$

where  $\bar{y}_{i,k}$  is the estimate of measurement  $i$  for frame  $k$  based on the current state estimate and the measurement function  $h(x)$ ,  $J_{i,k}$  is the Kalman gain for this measurement,  $C_{i,k}$  is the Jacobian of the measurement function  $h(x)$  and  $y_{i,k}$  is the measured value for this update. Effectively, the Kalman gain determines the 'ratio' of measurement estimate variance (by state) to measured-value measurement variance, and hence determines how much emphasis the filter should pay to the new measurement compared to its current state estimate. Consequently high variances in  $P$  lead to more emphasis on the measured values, while high measurement variance  $R$  leads to less emphasis on this measurement.

We use the following seven-state filter, maintaining the  $x$  and  $y$  positions of the robot (centred at the base of the robot's neck), its orientation, and the absolute  $x$  and  $y$  positions and velocities of the ball.

$$X = \begin{bmatrix} x \\ y \\ \theta \\ x_b \\ y_b \\ v_x \\ v_y \end{bmatrix}$$

## 4.2 Time Update

Each frame we update our estimated position based on odometry

$$\begin{bmatrix} x \\ y \\ \theta \\ x_b \\ y_b \\ v_x \\ v_y \end{bmatrix}_k = \begin{bmatrix} x \\ y \\ \theta \\ x_b \\ y_b \\ v_x \\ v_y \end{bmatrix}_{k-1} + \begin{bmatrix} d_f \cos(\theta) - d_l \sin(\theta) \\ d_f \sin(\theta) + d_l \cos(\theta) \\ \phi \\ \frac{v_x}{30} \\ \frac{v_y}{30} \\ 0 \\ 0 \end{bmatrix}_{k-1}$$

where  $d_f$  is the forward motion,  $d_l$  is the left motion, and  $\phi$  is the anti-clockwise rotation of the robot. We defined left, forward and anti-clockwise as positive because this is how the robot's sensors are defined, as well as the orientation of planar geometry. We add one thirtieth of the ball's directional velocity to the estimate of it's position as velocity is measured in cm/sec, while the update is performed once per frame, or thirty times a second.

The above function leads to the following Jacobian:

$$\begin{bmatrix} 1 & 0 & -d_f \sin(\theta) - d_l \cos(\theta) & 0 & 0 & 0 & 0 \\ 0 & 1 & d_f \cos(\theta) - d_l \sin(\theta) & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & \frac{1}{30} & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & \frac{1}{30} \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The non-zero non-diagonal elements of this matrix cause a correlation between the position states and the orientation, an between the position and velocity estimates for the ball. In fact, it is this correlation that leads to an estimate for the velocity, through the Kalman Filter update algorithm, on ball-distance and -bearing updates. This velocity is decayed by 5% for each frame that the ball is not seen, so that poor estimates of the shared-ball don't unduely influence teammate's estimates of the balls position.

The model covariance matrix Q is a diagonal matrix that drifts estimate variances higher to increase the effect of measurements when no objects have been seen for

some time, as well as ensuring numeric errors do not occur.

### 4.3 Measurement Update

Each time fixed object on the field (beacons, goals and field-line intersection points, known as corner-points), as well as the ball, is seen, a measurement update is performed in the Kalman Filter for distance and bearing to the object. The distance update is based on the horizontal distance from the base of the robot's neck to the centre of the object

$$y = \sqrt{(x - x_i)^2 + (y - y_i)^2}$$

and bearing to an object is

$$y = \tan^{-1} \frac{y - y_i}{x - x_i} - \theta$$

The standard error for each measurement is calculated by

$$s.e. = \sqrt{C_{i,k} P_{k|k-1} C_{i,k}^T + R}$$

For most objects, when a measurement is more than five standard errors from the state-based estimate of the same object, that measurement is marked as an outlier, and is not used in updating the Kalman Filter estimates. In this case we assume that the object we thought we saw was not in fact the true object but something else the vision system mistook for the object, and so we should not base our position on such estimates.

On the whole all objects are processed in the same manner in measurement updates. Three main differences occur. The first is that outlier rejection is not used on the ball. The reasons for this are two-fold. For one, measurements for the ball can be somewhat poor at a distance, so we don't want to reject reasonable measurements because we believe them to be an outlier. In addition, if a ball is not seen for sometime, but we receive poor shared-ball estimates, our estimate of the ball's position might not be correct, and we may ignore good ball measurements in favour of our poor prior belief. The second difference is based on the idea that we prefer goal posts, which are easily located, to goal centres, which are much more subjective. Consequently, any time we see a goal with at least one reliable post (that

is, a post we can reliably determine the distance to) we use that rather than the centre of the goal. However in the case where we don't have a reliable post we fall back on the goal centre. The third difference is in corner points. For most objects we use both distance and bearing measurements, and base the measurement variance  $R$  on sound theory which is then tuned to optimise performance. However because of the nature of corner points distance to them is quite hard to calculate, and even then can be quite inaccurate, particularly at distance. Because of this we inflate the distance variance of corner points to such a level that the standard error is as high as 10000cm, so that effectively that measurement is ignored. Corner point bearings are, however, quite reliable, and so are treated the same as all other objects.

A very accurate measurement can be calculated when two fixed objects are seen on the field, which we have coined Two-Object. Put simply, while bearing to an object is heavily dependant on your own orientation, the angle induced at the robot by the two objects is not, and so has a much greater effect on the position of the robot. We calculate the cosine of the angle reported by the vision system, and perform a measurement update on this compared to our estimate of what the angle should be based on the robot's estimate of it's current position.

#### 4.4 Filter Reset

Although the Kalman Filter is very accurate, it is far from perfect, and when a series of poor information, either vision or odometry, is received, without contrary 'good' information to correct it, the robot's estimates of it's states can become quite biased. When this occurs good measurements might be thrown out as outliers, and so the robot will not correct it's poor estimates. To ensure this doesn't happen we take note of any object that has a measurement rejected, and when a series of constraints (listed below) are met the filter resets, setting the covariance matrix back to it's initial (extremely high) variances, meaning that when the next measurement is processed the filter will pay almost no heed to it's current estimates, basing the next estimate almost entirely upon the new measurements. The constraints are that at least 2 objects must have an object-error greater than 0.3 in that frame, and the object-error-sum must be greater than 3. This is the sum of the errors for each object. Each object stores it's current error-sum, which is incremented every time that object has it's distance or bearing rejected as an outlier, and it is decayed by 6%

each frame. This allow objects rejected previously to have an impact on resetting this frame, while not allowing such impact too long after the rejection.

#### 4.5 Clipping

The Kalman Filter is a state observer that selects the optimal state for the information it has been given. However in some cases poor data can lead to 'impossible' states being developed in the filter, such as the robot being positioned well off the field. Simply restricting the state vector to being within the appropriate area after each operation is a solution to this problem, but does not take full account of the filter's knowledge of the probability distribution of the states. Instead we move along the ellipse of maximum probability until we are within the allowable range of values. For example, if we are outside the field past the back of the blue goal ( $x > 300$ ), we apply the following correction to the state vector  $X$ , using the covariance matrix  $P$ :

$$v = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$X_{new} = X - Pv(v^T X - 300)/(v^T Pv)$$

A similar method can be used to restrict position and orientation based on visual information. We use this to restrict orientation of the goalie when it can see it's own goal. Obviously any sort of information-dependant clipping can have a large effect on our state estimate, so we need to make sure we are basing our correction on reliable data. We do this by only applying the correction when we see our own goal at less than 100cm away, indicating a blob at least 70 pixels in height and 10 in width, which is unlikely to occur as noise. If we can see the goal at this distance it indicates that our orientation (including bearing to the goal) is within  $\frac{\pi}{2}$  of backwards, and the following clipping algorithm is applied:



$$v = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$X_{new} = X - Pv(v^T X + \phi - \frac{\pi}{2})/(v^T Pv)$$

where  $\phi$  is the bearing of the goal from the robot.

#### 4.6 Shared Ball

In order to operate effectively it is essential that each robot in the team has a reasonable estimate of the relative position of the ball. In order to accommodate this when one robot cannot see the ball, each robot passes to each other it's estimate for the  $x$  and  $y$  coordinate of the ball (the 4th and 5th states in the EKF) along with the standard deviations of these estimates, and these are used as additional measurement updates for robots who cannot see the ball in the last three frames. In initial tests we used the true variance of these estimates as measurement variance, but this led to too heavy a focus on poor ball measurements, particularly in the case of false balls seen by other robots, so now we double the variance estimate.

## 5 Locomotion

The locomotion engine was completely re-developed in 2005. It was initially developed in Python with only the inverse kinematic mathematics in C++.

Due to some last minutes processing requirements (two weeks before robocup) most of the walk engine was ported to C++. The main control remained in Python but called out to C++ for all the computational work. Since the port happened so late our 2005 code has the ability to switch between Python and C++ on the fly. This enabled us to prototype changes in the mathematics using Python.

### 5.1 Walk Engine

The principle of the walk engine is the same as in previous years, that is an omni-directional parameterised walk. We will refer you to our previous reports [Bunting *et al.*, 2003; Quinlan *et al.*, 2004] and the original omni-directional walk paper [Hengst *et al.*, 2002] for more information.

The major change this year was the addition of individual leg control. In the past the walk engine has contained parameters that control the both front and both rear legs of the robots.

Substantial time was also put into forward kinematics required to calculate the stance of the robot. The offsets needed in both the forwards and inverse kinematics, were tuned aswell as the inverse kinematics being modified to stop mathematics breakages. For example if the paw is told to move outside its physical range of the leg then the paw position is projected inwards and if the paw position is inside the physical range then the paw point is project outwards.

The walk used at Robocup 2005 was approximately 42 cm/s. This speed did depend on the robot. Throughout the year we generated many different walks with similar speeds, but the one used was chosen because of its ‘flat’ body and its apparent power (ability to move while being bumped).

#### 5.1.1 Individual Leg Control

The idea behind individual leg control is to improve the robots flexibility. For example kicks can now be built using standard walk commands. It is also beneficial for controlling the ball, when approaching the ball the left leg can be moved out

slightly if the ball is left of centre. It has also been noted that you can turn faster if you move legs in opposite directions. Another situation is walking in an arc, you can choose to take a smaller step on the inside leg (or move the inside leg slower). Although these changes seem small they greatly improve the flexibility of the walk engine.

We now allow the speed of each leg to be different, i.e the left front leg can be moving at twice the speed of the right front leg. This means that we need to re-sync the legs when a normal walk command is issued. Our code has built in support to re-sync all four legs, or re-sync in pairs (front or rear), meaning the behaviour developer does not need to worry about synchronisation issues.

## 5.2 Head Control

Head control can be broken down into two areas: General movements and Tracking an object.

### 5.2.1 General movements

General movements include searching for balls, panning for beacons and any other action that does not actually involve reacting to vision. The movements are generally straight forward but we have found that two common traps occur:

**Speed** Often the head is told to move too fast, this blurs the image. Additionally you need to match the head speed with the body speed. For example, if you are searching for a ball by spinning on the spot, the head movement must be fast enough that can not turn past the ball before the head travels its full path (no blind spots).

**Head height** By carefully setting the height of your head you can greatly decrease the amount of noise introduced into vision. For example, you won't see people wearing orange shirts if you don't look up into the crowd. We spend time making sure our head movements are as low as possible (yet high enough to see the beacons and goals) during games.

### 5.2.2 Tracking an object

Controlling the head to focus on the ball or another object is a visual servoing problem. It is therefore natural that we turn to this field in looking for the most efficient way to control the robot's head. We define the angle of the pan motor by  $\theta_{\text{pan}}$ , and the two tilt motors by  $\theta_{\text{big}}$  and  $\theta_{\text{small}}$ . The “small” tilt motor is the one that pivots inside the head of the robot, while the “big” tilt motor is the one that pivots inside the body of the robot. In general, the true head angles,  $(x_{\text{pan}}, y_{\text{tilt}})$  will be related to the joint angles by a nonlinear equation encapsulating the forward kinematics of the robot neck:

$$(x_{\text{pan}}, y_{\text{tilt}}) = \underline{f}(\theta_{\text{small}}, \theta_{\text{big}}, \theta_{\text{pan}}) \quad (1)$$

Note that to avoid doing on-line inverse kinematics, and also since we need to solve the problem of actuator redundancy, we convert to a small deviation model of (1). The change in the three joint angles is denoted by  $\delta\theta_{\text{pan}}$ ,  $\delta\theta_{\text{big}}$  and  $\delta\theta_{\text{small}}$ . We model the change in the effective angle of the head - in terms of pan ( $\delta x$ ) and tilt ( $\delta y$ ) - using the Jacobian of  $\underline{f}$  as follows:

$$\begin{pmatrix} \delta x \\ \delta y \end{pmatrix} = \begin{pmatrix} 0 & \sin(\theta_{\text{pan}}) & 1 \\ 1 & \cos(\theta_{\text{pan}}) & 0 \end{pmatrix} \begin{pmatrix} \delta\theta_{\text{small}} \\ \delta\theta_{\text{big}} \\ \delta\theta_{\text{pan}} \end{pmatrix} \quad (2)$$

For brevity, we define the Jacobian matrix  $\mathbf{X}$  as:

$$\mathbf{X} = \begin{pmatrix} 0 & \sin(\theta_{\text{pan}}) & 1 \\ 1 & \cos(\theta_{\text{pan}}) & 0 \end{pmatrix} \quad (3)$$

In order to aim the head at a particular point, we determine the distance in degrees that the head must move in each axis in order to aim at the point - that is, we determine a  $\delta x^*$  and  $\delta y^*$ . Given these target angle changes, we must determine how to drive the head motors to aim at the desired location (i.e., obtain  $\delta\theta_{\text{pan}}$ ,  $\delta\theta_{\text{big}}$  and  $\delta\theta_{\text{small}}$ ). Clearly, we must solve (2) for  $\delta\theta_{\text{pan}}$ ,  $\delta\theta_{\text{big}}$  and  $\delta\theta_{\text{small}}$ , and where we set  $(\delta x, \delta y) = \alpha(\delta x^*, \delta y^*)$  with  $\alpha$  a positive real constant determining how rapidly we try to correct visual servoing errors.

Note that the matrix  $\mathbf{X}$  is not square so we cannot determine its inverse. Instead, we use the pseudo-inverse (denoted  $\mathbf{X}^+$ ), which can be calculated using:

$$\mathbf{X}^+ = (\mathbf{X}\mathbf{X}^T)^{-1} \mathbf{X} \quad (4)$$

Then assuming  $\alpha = 1$  (which our observations show to be both fast and stable), we have:

$$\begin{pmatrix} \delta\theta_{\text{small}} \\ \delta\theta_{\text{big}} \\ \delta\theta_{\text{pan}} \end{pmatrix} = (\mathbf{X}\mathbf{X}^T)^{-1} \mathbf{X} \begin{pmatrix} \delta x^* \\ \delta y^* \end{pmatrix} \quad (5)$$

This solution results in the head aiming in the correct direction using the smallest possible change in joint angles from the current location. However, direct implementation of (5) is limited since it does not limit the range of movement of the motors. This can give rise to problems such as the robot accidentally hitting the ball with its head. Fortunately, it is possible to modify the formulation above to lock certain joint angles at particular times.

### 5.3 Forward Kinematics

This year forward kinematics were used for the elevation check and horizon line calculations, therefore their accuracy was heavily relied upon.

The mathematics for the forwards kinematics can be found in [Röfer *et al.*, 2004]. We felt some of the offsets used are debateable so we derived/measured these offset ourselves.

$$\begin{aligned} \text{top length of the leg} &= \sqrt{69.5^2 + 9^2} \\ \text{bottom length of the front leg} &= \sqrt{28.3^2 + 71.5^2} \\ \text{bottom length of the rear leg} &= \sqrt{21.3^2 + 76.5^2} \\ \text{rotator offset} &= 0.1286 \text{ radians} \\ \text{front knee offset} &= -0.1709 \text{ radians} \\ \text{rear knee offset} &= -0.2606 \text{ radians} \end{aligned}$$

The important part of the forwards kinematics is calculating the contact point on the ground. The nature of the Aibo walk means that the front paw does not touch the ground, but rather some point of the curved forearm touches the ground. Accurately determining this point is important when calculating the tilt and roll of the robot.

To simplify matters we did not implement the equation of the arc, but rather we calculated the height at certain intervals on the arc. The point with the lowest height (below the shoulder) is deemed to contact the ground.

The same process is repeated on the rear legs. In both cases the 'forearm' contact point is compared to the paw height, the lowest of the which is deemed to be the contact point (this takes care of the case where the paw is touching the ground).

## 6 Team Behaviour

Every year the importance of team behaviour is growing, this is partly due to increased parity in the low level skills. This year the NUbots took a more decentralised approach to team behaviour, all decision making is made by each robot with no negotiating of any type. The other new addition is potential fields (both potential points and lines of potential) being used to improve the adaptability of the behaviour. Although we don't directly adapt to an opposing team the potential fields do indirectly shift robots around depending on the state of the game.

All of the team behaviour was written in Python, hence no code could be salvaged from previous years. Many parts of team behaviour were also developed and tested in our simulator.

### 6.1 Positions / Roles

Early in the development process it was decided that the robots should choose their positions independently. That is, there would be no "captain" robot who would decide positions and communicate it to teammates. Using a captain robot can potentially prevent robots from taking on the same position and thus crowding parts of the field. If something happens to the captain, however, the decisions must be made by another robot. The complexity of creating the necessary redundancy appeared to outweigh the advantages of having a captain.

On the field each robot, apart from the goalkeeper, can take on any of the positions. The position they take on is dependent on the formation the team is currently in, and their proximity to the positions in the formation.

#### 6.1.1 Formations

The formation of the team is decided by each robot individually. In the RoboCup 2005 competition only very few formations were used, and these were very simple. If the ball was in the opposition half then the attacking formation is used. Two robots take the left forward and right forward position and one stays in the defensive half in line with the ball. If the ball is in the defensive half then two robots will take centre back and left or right back positions depending on where the ball is and one stays in the opposition half in line with the ball. For the semi-final and grand-final an

additional formation was used. This formation has a sweeper who tries to position themselves just outside of the penalty area and between the goal and the ball. This formation is used when the ball is in the defensive half between the half-way line and the beacons. Additionally a centre back will stay further up the field and one robot will be in the opposition's half.

### 6.1.2 Positions

All positions are defined by coordinates and robots in these positions move to the coordinates of the position after it has been acted upon by the potential field.

The current positions are goal keeper, centre back, left and right back, centre forward, left and right forward, and sweeper. Coordinates are defined for starting with the kick off, starting on the opposition's kick off, and general field play.

During field play robots decide the formation and then calculate their distance, and their teammate's distance, to every position in the formation. Deciding which robot is assigned to which position is done by solving the problem:

$$\text{minimise } \sum_{i \in R} \sum_{p \in F} d(c_i, c_p) x_{i,p}$$

$$\text{subject to: } \sum_{p \in F} x_{i,p} = 1, \forall i \in R \quad (1)$$

$$\sum_{i \in R} x_{i,p} = 1, \forall p \in F \quad (2)$$

$$x_{i,p} \in \{0, 1\} \quad (3)$$

where  $c_i$  is the coordinate vector of robot  $i$ ,  $c_p$  is the coordinate vector of robot  $p$ ,  $d(x, y)$  is the euclidean distance between two coordinate vectors, and  $x_{i,p}$  is the binary variable which is 1 when robot  $i$  is assigned to position  $p$  and 0 otherwise. If a robot is currently chasing the ball, i.e. they are the closest to the ball, they have priority when being assigned a position. Since they will hopefully be getting possession of the ball, they take on the position closest to where the ball is.

## 6.2 Potential Fields

This year we added a potential fields to control some of the movements of the robots. In our system only off ball positioning is controlled by the potential field.



We do not use the potential field for path planning, as result we are only interested in the potential at the robot at the present time. This allows us to save on computation as the gradient of the potential at this point gives us the direction and size of the movement required.

In our system we have two types of attractors/repulsors : those being points and lines. In both cases the attractiveness/repulsiveness is modeled under a normal distribution and controlled by two parameters,  $\sigma$  (spread) and  $\alpha$  (height). A negative  $\alpha$  implies a repulsive point or line.

### 6.2.1 Points

Points are common in most potential field systems and are used for object such as other robots, the ball, the goal etc. In our system we model: *ball*, *homePosition*, *keeper*, *team1*, *team2*, *team3*, *ownGoal*, *oppGoal*, *closeball*. The home position is that decided upon in the formation and position subsections above. In general we are attracted to the ball, attracted to your home position, repulsed from your keeper and team mates (keeper is treated separately as we like to give it more space), repulsed from our own goal as this stops robots becoming illegal defenders when not chasing the ball. Opponent goal and close ball are only used when required, i.e. when dribbling the opponent goal is an attractor and if the robot is a keeper the close ball is also attractive.

You may wonder why the ball is included twice for the keeper. Obviously you do not want your keeper leaving the penalty area, so at a distance we only wanted the ball to have a slight influence, its is not until the ball is close to the penalty area that we want the keeper to move, this is achieved by the close ball attractor.

The ability to have multiple attractors/repulsors for each object is interesting, you can create situations where you are attracted to an object until you get too close and then repulsed from it. This gives you the ability to quickly create rules like ‘move close to but not too close to an object’. Many complex rules can be created quickly using sums of attractors on a given objects.

We had intended to add attractive points to the the ‘boy’ ball replacements points, this was not done in time for Robocup 2005.

### 6.2.2 Lines

To our knowledge we are the first team to use potential lines. These are very powerful, and greatly assist in the generation of rules. Unfortunately we did not get the chance to incorporate all the intended features of the lines into our 2005 code.

We did however put in repulsive lines for all edges of the field, by having a low  $\sigma$  and high  $\alpha$  you can make it impossible for your robots to drift off the field while positioning (barring a localisation mistake). This enables you to construct the other points of potential without giving thought to a robot leaving the field, i.e. if the robot is repulsed from the ball and the ball is nearing the sideline, then the robot will not walk off the field (30 sec penalty) to avoid the ball rather he will move in some direction (forwards or backwards) to avoid both the ball and the line. Lines of repulsiveness were particularly useful for controlling the dribbling behaviour.

We see lines being beneficial in the future for controlling passing or blocking/avoiding shots. For example, if a robot's teammate has control of the ball then an attractive line can be projected from the ball up the field in the direction of the robots heading. This would 'drag' teammates to be in line with the pass (or be in a better position to run onto a pass). This can also be used to position your robots to better block opponent kicks.

Repulsive lines can also be used to force teammates out of the way. e.g when taking a shot on goal, a robot can attempt to create a clearer path by making an repulsive line between himself and the goal.

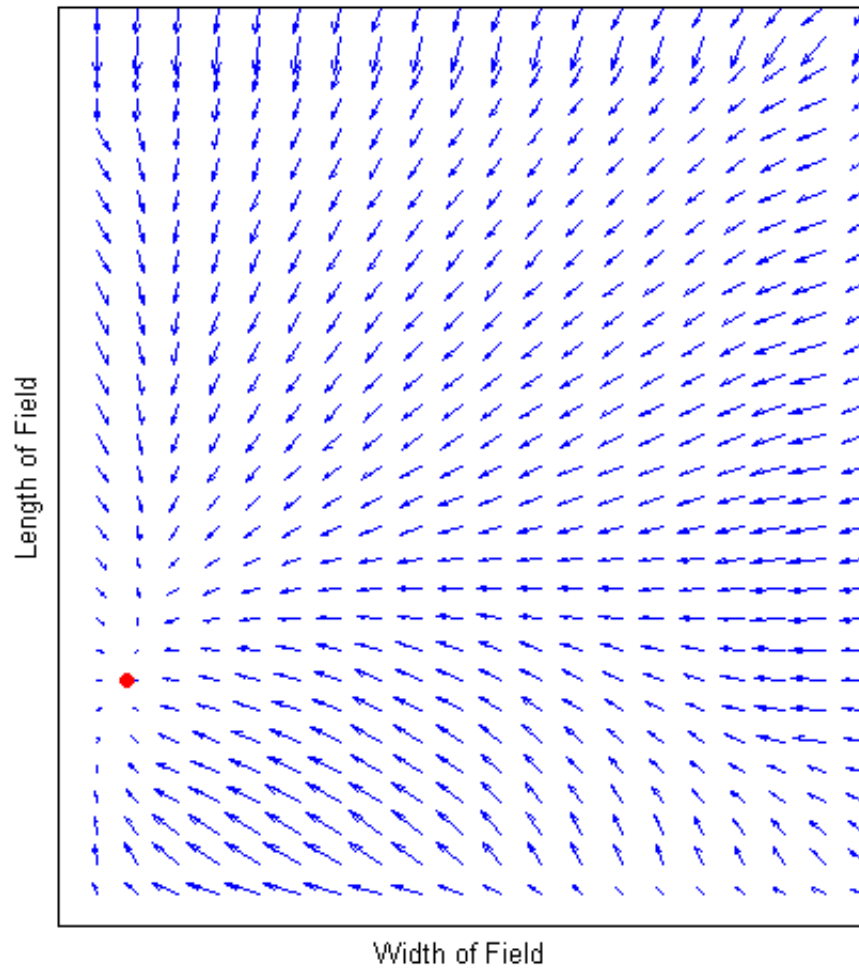


Figure 58: Potential field example. Arrows represent the desired movement at every point on the field. The robot has been instructed to leave the field, but the repulsive lines are keeping the robot in field of play (its final destination is shown with the red dot). You can also notice the effect of the field pulling the robot around its own penalty box (the goal would be at bottom centre of the image). The arrows on the right go up and around the penalty area, while those inside the area quickly move the robot away from the box)

## 7 Individual Behaviours

The individual skills and behaviours of the robots are arguably the most critical element of the game play. The importance of developing and fine tuning these skills should not be underestimated.

All of the individual behaviours were written in Python.

### 7.1 Chase

The NUBot philosophy has always been that to succeed at soccer you must quickly and confidently gain control of the ball. As a result a lot of attention is placed into to developing code for chasing and grabbing the ball.

It turns out that good quality chasing and grabbing relies on a high performance of all other modules. Accurate ball distances are required from vision, correct ball velocities from localisation and precise movement and odometry calibration from locomotion.

The development of our chase code this year was performed under a primary directive, that being NO ARTIFICIAL SLOW DOWN. In the past most teams have compensated for hidden defects in vision, locomotion etc by introducing small ‘tentative’ steps when close to the ball. We made the decision this year that we would NEVER introduce these steps and this would force us to fix the underlying problems. This decision was difficult for many new members to fully understand (i.e. why were we fumbling badly, when we could just slow down?), but it was critical in the development of the chase and grab code we used at Robocup. Our ability to run at full speed on to the ball and grab accurately (at least as accurately as opposing teams) was a critical reason why we won many games. In our eyes this was a key advantage we possessed over rUNSWift in the semi-final.

New in chase for 2005 is the ability for a robot to walk backwards or do a number of fast short steps to align themselves with the ball. This ability was tightly bounded, this avoided any unnecessary slowdown but allowed the robot to grab a ball that it previously would have missed

The basic chase code forces the robot to take the largest possible step which will position the ball on the chest -

```
distance = ball.distance
```

```

bearing = ball.bearing
relBallX = fabs(distance) * sin(bearing)
relBallY = fabs(distance) * cos(bearing)

strideLength = relBallY * 10.0
strafe = CROP(relBallX * 10.0, -25, 25)

frontStrideLength = CROP(frontStrideLength, -BestFrontStrideLength,
                          BestFrontStrideLength)
rearStrideLength = CROP(rearStrideLength, -BestRearStrideLength,
                        BestRearStrideLength)

maximumTurn = 0
sqr = sqrt(rearStrideLength *
           rearStrideLength / BestRearStrideLength + (strafe^2)/50.0)
if (sqr < 50): maximumTurn = 50 - sqr
absMaximumTurn = math.fabs(maximumTurn)

turn = RAD_TO_DEG(ballHeading)
turn = CROP(turn, -absMaximumTurn, absMaximumTurn)

```

The ability to go backwards or take small strafe steps was based around a simple idea - “If the ball is reasonably close and not in-line with the chest, then quickly adjust until the robot can continue its normal chase”. Obviously the actual numbers need to be tuning based on the performance of the vision, localisation and locomotion modules. At Robocup we had the following code -

```

if (relBallY < 50 and fabs(relBallX) > relBallY/5.0):
    relBallY = relBallY-10
    <adjust stance slightly to one that worked
    better at a higher step frequency>
    strideLength = (relBallY) * 10.0

```

It is important to note that the existence of these steps is all but hidden from a human spectator. The steps are subtle in nature and often less than one full step is required to realign the chase.

### 7.1.1 ‘Avoiding’ a robot while chasing

The intention of the avoidance code is not to stop and completely avoid the robot but rather ‘edge’ around the obstructing robot. The forward component of the chase is not modified, only a sideways (strafe) component is added. In practice this results in an arcing run that just avoids (or sometimes brushes) against the obstructing robot. This movement has a very minimal effect on the speed of the chase, but greatly reduces the number of collisions that occur when chasing.

```
if (otherRobot.seen):
    robotDistance = otherRobot.visionDistance
    if (ball.distance < robotDistance-10): return 0
    robotBearing = robot.visionBearing
    if (robotBearing < 0) :
        strafe = 50
    else :
        strafe= -50
```

## 7.2 Grab

Grabbing can be considered as a form of the *chicken and egg dilemma*, in the respect that to play a ‘grab and move’ style requires good grabbing, but good grabbing is only important in a ‘grab and move’ style.

Ever since 2002 we have employed a ‘grab and move’ style, so for our team the grab is extremely critical. Efficient grabbing relies on everything working well, small errors in any module will severely effect performance. Any changes made in other modules effect the grab, e.g. if ball distances are improved then the ‘magic’ numbers developed are no longer correct and need to be retuned.

This year was the first time we had a ball velocity and used the shared ball for grabbing. As a result the localisation module was heavily relied upon.

For Robocup an immense amount of time and effort is placed into developing and maintaining the grab code.

### 7.2.1 Grab Trigger

The point where a robot needs to trigger a grab is on a never ending cycle of adjustment and tuning. This year saw a slightly different approach taken, for the first time we implemented sanity factors. In this method multiple grab triggers can exist with a weighting applied to each one. Our final version relies upon the relative location of the ball, the velocity of the ball and the tilt of the head.

```

distance = ball.distance
bearing = ball.bearing
relX = fabs(distance) * sin(bearing)
relY = fabs(distance) * cos(bearing)
velX = ball.vX      # velocity X
velY = ball.vY      # velocity Y
tilt = RAD_T_DEG * (Sensors.S_HEAD_TILT_BIG+Sensors.S_HEAD_TILT_SMALL)
grab = False

velY=CROP(velY,-20,20)
grabSanity = 3.0 * fabs(relX) + (1+(velY/100)) * fabs(relY)
if (tilt < - 30): grabSanity += 2
elif (tilt < - grabTilt): grabSanity -= (3 * (tilt+30))
elif (tilt < - 30): grabSanity -= (tilt+30)
if (grabSanity < 20):
    grab = True

return grab

```

### 7.2.2 Grab Trigger - Learning

At one stage a simple neural network was created as an attempt to learn the function associated with the sanity factors. The robot would kick the ball a short distance and then chase and grab the ball. At this point the robot could test to see if the grab was successful and a simple ‘yes’ or ‘no’ reinforcement was given to the network.

Unfortunately the training took place when ball velocities supplied by localisation were wrong and thus the resultant network only performed marginally better than the original hand-tuned numbers. Ball velocities were only fixed in the last

days before Robocup, hence the learning was not re-tried and hand-tuned values were used instead. For 2006 some variety of learning will definitely be implemented.

### 7.2.3 Grab Motion

This years grab motion relied heavily on localisation, we would walk in the direction of the shared ball while doing the grab. The benefit being that grab motion itself would try and cover up an error in the grab trigger, it also allowed us to handle balls that were rolling away or towards the robot.

The 2005 grab was our fastest grab ever, the entire motion was completed in at most 10 vision frames ( $\leq 1/3$  of a second). We would then transition from the grab into either a ‘dribble’ or a kick, the speed of the grab enabled over 2 seconds of ‘dribbling’ time.

The grab could be completed faster if the ball was found to be on the chest of the robot, one frame would be removed from the grab for every frame that the ball was on the chest (leaving a minimum grabbing time of 5 vision frames or  $1/6$  of a second).

During the grab the head would first be lifted, allowing the ball to slide under and then the head would be dropped over the ball in conjunction with the mouth being opened. While the head is moving the front legs are moved slightly forwards and outwards to surround the ball.

```
tempX = (10 * fabs(ball.distance)) * sin(ball.bearing)+ball.vX/30
tempY = (10 * fabs(ball.distance)) * cos(ball.bearing)+ball.vY/30
forwards = CROP(tempY,50,BestFrontStrideLength)
strafe = forwards,CROP(tempX,-30,30)
turn = 0.0
Walk(BestStepFrequency,forwards,strafe,turn)
if (Sensors.S_INFRARED_NEAR<DRIBBLECANCELIR+1000):
    stateCounter++
if (stateCounter < 2):
    SetHead(55,-15,0)
    leg[0].forwardOffset = WalkData.DefaultFrontForwardOffset+7.0
    leg[0].sideOffset = WalkData.DefaultFrontSideOffset+5.0
    leg[1].forwardOffset = WalkData.DefaultFrontForwardOffset+7.0
```



```

    leg[1].sideOffset = WalkData.DefaultFrontSideOffset+5.0
else:
    mouthPosition = -55.0
    leg[0].forwardOffset = WalkData.DefaultFrontForwardOffset+15.0
    leg[0].sideOffset = WalkData.DefaultFrontSideOffset+7.0
    leg[1].forwardOffset = WalkData.DefaultFrontForwardOffset+15.0
    leg[1].sideOffset = WalkData.DefaultFrontSideOffset+7.0
    if (stateCounter > 7):
        Head.SetHead(30,-60,0)
    else:
        Head.SetHead(50,-60,0)
if (stateCounter > 9):
    WalkData.mouthPosition = -55.0
    return 0    # grab done
stateCounter ++
return 1      # continue

```

### 7.3 Dribbling (“Holding”)

One of the key features of our play this year was the omni-direction dribble employed. This dribble was primarily used to line up the goal or to avoid other robots. It should be noted that many of the ‘nice’ features of our dribbling (i.e. dodging robots, kicking through gaps etc) are not the result of overly advanced vision but rather a combination of simple but deliberate rules (albeit these rules required a lot of effort to create and tune).

The stance used for the chase had to satisfy multiple conditions -

**Omni-directional** - For optimal use the stance should give full omni-directional movement, this allows the robot to strafe around robots, dribble between opponents or back the ball off sidelines.

**Control the ball** - Obviously the robot should never drop the ball but finding the right combination between omni-directionality and holding was difficult. We found the optimal solution was to have a ‘loose’ hold on the ball, that is the ball could bobble between the legs and the head but could never pop out.

**Vision Distance** - In our system the robot could see goals from over half the field while dribbling the ball. This enabled us to line up long range shots or avoid distant robots.

Walk optimisation was then run to improve the speed of the dribble. In these experiments the robot ran between two pink balls while dribbling the orange ball. Some parameters such as `forntForwardOffset` and `frontSideOffset` were constrained to make sure the ball was held and the final parameters allowed the robot to dribble at approximately 35cm/s

An internal timer was used to monitor the dribbling time, it forced the robot to cancel before it got close to the 3 second limit.

The dribble logic can be summarised as *Check for ball* then *dribble away from sidelines* or *dodge based on vision* or *dodge based on infrared* and then *line up the goal*. Full code for this is shown below, `FinishDribble(int)` checks to see if the dribble is over the time limit and `object.visionOrientation` returns true if the object is in the centre of the image. Further information on each step can be found in the following subsections, this code block is presented to show the full sequence of decision making.

```
#Check for ball
if (timeDribbled > 10 and Sensors.S_BODY_PSD < DRIBBLECANCELIR):
    return -1    # Ball not found, quit dribble
mouthPosition = -55.0
SetHead(55,-53,0)
Vision.CheckForObstruction()

goal = OPPONENT_GOAL
goalGap = FO_OPPONENT_GOAL_GAP
# If we begin the dribble near the edge of the field walk away
# from the sideline for a step or so (max of 20 frames)
if (fabs(me.y)>160 or (fabs(me.x)>250 and goal.seen == 0)):
    if (DribbleOffLine(20)==1):
        return FinishDribble(timeLeft)
    else:
        timeSpentBackwards = 100;
```

```

if ((fabs(me.y)<80 and fabs((RAD_T_DEG*me.orientation)) < 30)
    or fabs((RAD_T_DEG*goal.bearing)) < 30):
    if (Dodge() == 1):    # Dodge Done
        return FinishDribble(timeLeft)
    elif (DodgeIR() == 1):    # Dodge Done
        return FinishDribble(timeLeft)
lineUpResult = LineUpGoal()
if (lineUpResult == 1): # Still trying to line up goal
    return FinishDribble(timeLeft)
if (lineUpResult == 0): # Goal lined up now do something else
    if (me.x > -100):
        #check that goal is very very lined up !
        if (goalGap.visionOrientation == 1 and goalGap.seen):
            timeLeft -=5
            DribbleToTarget(goalGap)
        else if (fabs(me.y)>60 and me.x > 120):
            DribbleToTargetWithVeerToCentre(goal)
        else:
            DribbleToTarget(goal)
    else:
        timeLeft -= timeLeft #can dribble a bit but would prefer to just kick !
        if (Vision.middle < 150):
            turn = RAD_T_DEG*goal.bearing
            if (turn > 0): turn = 30
            else: turn = -30
            DribbleToTargetWMWithTurn(goal,turn)
        else:
            DribbleToTarget(goal)
    return FinishDribble(timeLeft)

return FinishDribble(timeLeft)

```

### 7.3.1 Lining up to shoot at goal

If the goal is visible then turning towards it is relatively straight forward, if the goal is not visible then one must rely on localisation. The important part is that you can construct a more reliable rule for facing the opponent goal than simply turning towards the opponent goal. At first glance this statement may not make much sense, but it is often better to turn away from your own goal. In most situations turning away from your own goal is equivalent to turning towards the opponent goal, but cases do exist where they are not one and the same. Our logic for choosing the turn direction is shown below -

```
angleToTurn = -ownGoalAngle
# Own goal and oppoent goal are in the same direction
if (fabs(ownGoalAngle) > 0 and fabs(opponentGoalAngle) > 0):
    if (me.x > -100): # If attacking .. turn towards opponent goal
        angle = opponentGoalAngle
    else: # If defending .. turn away from own goal
        angle = -ownGoalAngle
```

Our robots will then turn until they see the goal or expect to see the goal based on localisation. If the goal is seen then they line up to the goal gap not the actual goal. In our Robocup code the robot would continue turning in the same direction if the the robot expected to see the goal but didn't see the goal. At various stages we experimented with fully trusting localisation, but too often this resulted in the robot finishing its turn just before the goal would have become visible.

Once lined up we have two main options, the robot can kick the ball or the robot can dribble the ball to a better position to kick.

When attacking we tend to dribble the ball a bit longer in an attempt to further line up the goal, this can be seen in the following code (a subset of the full dribbling logic shown above).

```
if (goalGap.visionOrientation == 1 and goalGap.seen):
    timeLeft -=5
    DribbleToTarget(goalGap)
else if (fabs(me.y)>60 and me.x > 120):
    DribbleToTargetWithVeerToCentre(goal)
```

```

else:
    DribbleToTarget(goal)

```

The first *if* statement says that if the goal gap is in the middle of the image (i.e. straight ahead) then dribble towards it but quickly reduce the remaining dribble time (so to make sure we get a kick away before being tackled). The second *if* is triggered when the robot is in one of the attacking corners, in this case the robot dribbles towards the goal but also veers towards the centre of the field. The veer opens up the face of the goal, therefore creating a better target to shoot at.

If the robot is defending we typically want to quickly clear the ball but will also make an effort not to kick the ball into an opponent robot.

```

timeLeft -= timeLeft
if (Vision.middle < 150):
    turn = RAD_T_DEG*goal.bearing
    if (turn > 0):
        turn = 30
    else:
        turn = -30
    DribbleToTargetWMWithTurn(goal,turn)
else:
    DribbleToTarget(goal)

```

The first line quickly reduces the available dribble time, the *if* checks to see what is in the middle of the image (section 3.10), if the middle contains a possible obstruction then the robot will dribble towards but also attempt to turn and kick away from an obstruction but only towards the opponent goal. A decision was made that we would prefer our robots to kick into an opponent robot then kick the ball out. In practice this simple approach worked better then expected, with many clearing kicks being threaded through gaps.

### 7.3.2 Obstacle Avoidance - “Vision”

At Robocup 2005 vision based avoidance was only turned on for red robots. We continue to avoid a robot for the number of frames that the robot had been seen (up to 20 frames). This serves two purposes, it allows for a vision robot to disappear

for a frame but still be avoided but secondly it allows our robot to go around the obstacle once it has been avoided. The second point is critical, you *must* continue to avoid for frames after you have cleared the robot otherwise you often step back towards the robot (counteracting the initially good avoidance).

When using vision as the input we simply move away from the robot we are avoiding, once a direction has been chosen it can only switch if the bearing to the robot has changed significantly.

You should also note that we only avoid close robots (i.e those under 1 metre).

```

robotBearing = RAD_T_DEG*(robot.visionBearing)
robotDistance = robot.visionDistance

forward = robotDistance*0.5
if (robotDistance > 100): return -1
if (hasDodged == 0): lastRobot = robotBearing;
elif (fabs(robotBearing) > 20):
    lastRobot = robotBearing;
elif (hasDodged < 5):
    if (fabs(robotBearing - lastRobot) > 5):
        lastRobot = robotBearing;

if (lastRobot > 0):
    strafe = 50
    turn = 10
else:
    strafe = -50
    turn = -10
hasDodged =CROP(hasDodged + 3,1,20)
else :
    if (lastRobot > 0):
        strafe = 20
    else:
        strafe = -20
forward = 150

```

```
hasDodged =CROP(hasDodged - 1,0,20)
```

### 7.3.3 Obstacle Avoidance - “Infrared”

At Robocup the majority of avoidance was actually done using the infrared sensors as they work irrespective of lighting/vision conditions. It also has the added benefit of avoiding any obstacle, i.e. the goal post or a referees leg. The infrared avoidance works in conjunction with the vision avoidance, the *hasDodged*, *lastRobot* variables are shared so if oscillation between vision and infrared avoidance is seamless.

The first important step is tuning the infrared sensor value to a value that indicates an obstruction, the stance of the robot effects this number, ideally you want to maximise this value so that it does not cause the robot to avoid the ground (i.e. at what distance does the infrared cross the ground). We made the robot dribble with the ball and monitored the values of the IR sensors to discover this number. It turns out the sensors vary considerably between robots, so we had to customise this for each robot. The numbers for DRIBBLEDODGEIR varied between 200000 and 250000, this roughly equates to an avoidance distance of 20cm.

Deciding on the direction to avoid is hard since the IR sensors only give you a distance, if we have previously avoided on vision then we continue in that direction else we made the decision to always move towards the goal. This has the advantage of the robot never running outside the field and you increase your likelihood of scoring.

```
if (irDodge):
    obstacleDistance = Sensors.S_INFRARED_NEAR/10000.0
    forward = obstacleDistance
    if (obstacleDistance < 5): forward = -10
    if (hasDodged == 0 and hasIRDodged == 0):
        if (me.y >= 0): lastRobot = -50
        if (me.y < 0): lastRobot = 50

    if (lastRobot > 0):
        strafe = 50
        turn = 10
    else:
```

```

        strafe = -50
        turn = -10
        hasIRDodged =CROP(hasIRDodged + 3,1,20)
    else :
        if (lastRobot > 0):
            strafe = 20
            turn = 10
        else:
            strafe = -20
            turn = -10
        forward = 20
        hasIRDodged =CROP(hasIRDodged - 2,0,20)

```

As mentioned the IR sensor avoidance produces had an unexpected positive side effect. When dribbling near the base line the robot would trigger and avoidance on the goal post and would actually dodge the post and end up inside the goal. This meant our robots even when avoiding a keeper would always end up dribbling inside the posts as opposed to just outside the goals.

### 7.3.4 Avoiding sidelines

The key advantage of the omindirectional dribble was the ability for our robots to back the ball off any field lines. A great example of this is the last penalty kick in the final, where our robot dribbles off the line and spins towards the goal. Another good example of this was our last goal in the semi-final against rUNSWift, where our robot briefly backed a ball of the goal line (just outside the post) then spun and shot a goal between the post and the keeper.

We achieved this using our potential field system, the repulsiveness of the lines is turned up hence driving our robots away from the edges of the field. Using the potential field allows the robot to move smoothly, i.e in the corners of the field the robot will move diagonally out of the corner.

We did have to sacrifice some vision distance when moving backwards, the head needed to be placed further down to hold the ball harder.

```

WalkData.mouthPosition = -55.0

```



```

Head.SetHead(50,-60,0)
PField.SetDribble(True)
PField.SetHome(0.0,0.0)
PField.UpdateAttractors()
x,y = PField.CalculateXYShift()
PField.SetDribble(False)
Self = FieldObj.fieldObjects[FieldObj.FO_SELF]
DribbleToPoint(Self.x-x,Self.y-y,Self.orientation)

```

#### 7.4 Kick Selection

The effectiveness of our dribbling code simplified our kick selection logic. At Robocup we actually used 5 types of kicks, 4 of which went forwards at varying lengths, one went backwards but will not be spoken about due to the ‘kick out of the goal’ incident in the final.

Typically we tried to keep the ball in play at all times, hence we only used our most powerful kick in two cases: in the back of the defensive half and clearing to the front half or if the robot is lined up to the goal (thus it is a shot on goal). This powerful kick was similar to that used by UTS in 2004.

We used our second most powerful kick (sideswipe with the head that went forwards, which was actually a walk)) when we were roughly facing forwards but not going to score a goal. This meant we progressed the ball forwards but tried not to kick it out.

Our third most powerful kick is actually a dribble of the paw, it was used in a similar case to the second kick but usually when the goal can not be seen. It has the advantage of being extremely quick to execute.

The forth kick was more of a ‘drop’, it was used when we simply wanted to release the ball a small distance in front of us.

We also have ‘slap’ kicks that do not require a grab but they were not used at Robocup. This was not an entirely conscious decision, they were actually commented out to help test grabbing and we preferred the game play so left them commented out. Up until the days before Robocup the ‘slap’ kicks were in use.

We also have countless other kicks  $\approx 30$ , but they all have advantages and disadvantages. The fact that they often do not work on every robot led us to use only a

small percentage of them.

### 7.5 Moving to a Point

Moving to a point on the field maybe the most important act a robot can do. We have four ways to move to a point:

**Direct** Moves to a point (x,y) with no regards to orientation.

**Heading** Moves to a point (x,y,orientation). All three (x,y,orientation) are corrected at each point in the movement. A subset of this one is (x,y,FACEBALL). This is the most common movement as we usually want to face the ball when moving to a point.

**Heading - Forwards** Moves to a point (x,y,orientation). We first look at the point we are heading towards then run to the point and then turn to orientation.

**Heading - Backwards** Moves to a point (x,y,orientation). We first turn away from the point (x,y) then back up to the point. Once at (x,y) we turn to orientation.

Calculating the steps required to move to a point is either done using trigonometry or from the outputs of the potential field.

The quality of your localisation module is the sole influence on the performance of your move to point code.

### 7.6 Diving

The dive trigger has undergone many modification throughout the year. When the velocity calculations in localisation were fixed we settled on the trivial code below.

```
if (!ball.seen): return 0
ballX = math.sin(ball.bearing)*ball.distance
ballY = math.cos(ball.bearing)*ball.distance
if (ball.vY<0 and (ball.vY*-3 > ballY) and (math.fabs(ballX + ball.vX*3) < 60)):
    return DIVE
return NODIVE
```

In hindsight we probably dived a touch too often at Robocup, but the ability to have a quick trigger, in our case only one frame of vision because we used the shared ball, enabled us to make many quick and important dives.

One of the best dives occurred in the final when our goal keeper dived and saved a shot on goal, then the keeper simultaneously realised he had the ball and popped up and dribbled away to clear the ball. This showed the advantages of using one frame triggers (and the shared ball) for both diving and grabbing.

## 7.7 Unused Behaviours

During the development of the code we came up with substantial blocks of code that show great promise. For various reasons they were not used in games this year but should be incorporated into game code for next year.

### 7.7.1 “Paw” Dribbling

This was an extension of our 2003/04 ‘ramming’ code, where the robot would dribble (knock) the ball around using its paws. Both the German Team and CMU exhibited a similar type of behavior at Robocup (other teams may also have this code).

In our code you could aim the dribble by choosing which paw will hit the ball and also the paw offset on the ball. The offset allowed the robot to hit the ball at varying angles. This gave us the ability to dribble through gaps, off lines or score goals from angles.

“Paw” dribbling was developed as a backup in case we did not fix our grab. In future years it will be used to accommodate rules changes and also for its ability to speed up play (i.e. no grab required).

### 7.7.2 Controlled Diving

One other skill developed was a controlled dive, the principle here was that the robot would only dive the desired amount, effectively he would be ‘reaching’ for the ball as opposed to making a ‘kick’ style dive used by most teams.

Our new locomotion engine allowed individual legs to be moved, hence it was just a case of moving the desired paw to the point in the real world where the ball is (or will be). The advantage is that the rest of the robot can continue to walk, and the robot will only do a small move if required (hence making the recovery from

the dive faster). Another advantage is the fact that it tended to force the robot to catch the ball, i.e. as the ball hit the leg and started to move in towards the robot, the leg would simultaneously be moved in to match this new ball position and this would quickly move the ball under the chin.

This was not used at Robocup since it relied heavily on velocity prediction and sufficient testing time was not available.

## 8 Debugging and Offline Tools

This year we constructed a new vision application with embedded python. It runs the exact source code on the robot allowing for vision to be developed and tested offline. The general principle of the application is the same as those used in the past so it will not be further discussed. We also updated other applications that help in localisation and locomotion debugging.

The main new application was the simulator which will be discussed in this section.

### 8.1 Simulator

2005 saw the introduction of a new debugging tool to assist in the development of behaviours, a simulator dubbed NUsim. The objective of the simulator project was to build a system which would allow behaviours to be quickly prototyped. This was used extensively in the lead up to and the days of the competition.

By focusing on only the behaviours of the agents much of the complexity of the simulation can be removed. The returns afforded by a high fidelity simulation of the environment diminish since approximation of the low-level behaviours of the robots is sufficient. Inside NUsim the dynamics are crudely modeled starting with a perfect expectation of how they should occur. Random noise is then added to this model which approximate the deviations from expectations observed in the real world.

#### 8.1.1 Experiences

NUsim is the server in a client-server model. The client is a subset of the NUbots system which is run on the Aibo. After being integrated with the NUbots system NUsim was used to test the methods that calculated the movement of individual

robots. By including the existing model of robot movement in the NUsim world model it was possible to see the results of the new code instantly. Prototyping these behaviours in simulation not only provides savings in time and robot wear but also allow the output to be judged in a more objective way. At one time during the NUbots 2005 development there were problems with the vision and localisation modules which would cause inconsistencies with the Aibo's position on the field and its position in its world model. Using behaviours that are proven to work correctly in simulation it was then possible to identify problems in other modules.

Another example of a behaviour created with NUsim was the field positioning behaviour. The Aibos used their shared world model, with the coordinates of the other robots being broadcast to them, to calculate which position they should fill to minimise the total cost of the movement. These behaviours are quite difficult to test in the real world because of the dependance on the vision and localisation modules. Using NUsim and placing the simulated ball in different locations it is immediately obvious how the robots would try to place themselves in the real world. Changes can be made almost instantly and it is not necessary to transfer the new formations to each robot and then to wait for them move to their positions.

**Final Soccer Results:**

1. German Team (Germany)
2. NUbots (**Australia**)
3. rUNSWift (Australia)

## 9 Challenges

### 9.1 Open challenge

The open challenge attempted at Robocup was a joint challenge involving a pickup game with UNSW. The premise is that our robots can play a game of soccer with even though the two teams share no code apart from a predefined message protocol. This protocol was decided upon and was designed to be as general as possible (see below), no further communication was entered into, i.e. we did not discuss how our robots will interpret the messages or how our robots should behave on the field.

```
#ifndef PICKUP_H #define PICKUP_H
struct PickUpSoccerBroadcastInfo
    char header[4];    // PkUp
    int playerNum;      // 1-4
    int team;           // 0 is red 1 is blue

    // position of robot

    float posX, posY;  // same coordinate system as the    localization challenge
    float posh;        // radians, 0 along the +x axis, increasing counterclockwise

    // variance of position of robot

    float posVar, hVar;

    // position of ball

    float ballx;
    float bally;

    // variance of position of ball

    float ballvar;
```

```

    // destination of robot

    float destx, desty;

    // If see the visual ball.
    bool ballSeen;
;
#endif // PICKUP_H

```

From our point of view the major difficulty stemmed from the fact that we don't have the notion of a 'destination' in our code. We ignored these variables when a packet was received from rUNSWift and always sent (0,0) as our destination, how this effected rUNSWift is unknown.

For the challenge we played two of our robots and two of rUNSWift's robots against four German Team robots. After 6 minutes the game was still 0-0. During the game we observed position swapping, role negotiation and other features of advanced shared team behaviour.

It was noted that our robots tended to be more aggressive in attacking the ball then rUNSWift, this may be due to the fact that we didn't send a destination or simply due to the fact that we play a slightly more aggressive style. We tend to err on the side of making sure a robot is always chasing the ball rather than not chasing.

## 9.2 Variable lighting challenge

Our approach to solving the variable lighting challenge was to change the camera settings on the fly to adjust for changes in light.

The average value of the Y component in the image indicates the lighting level of the field. By setting the shutter speed in accordance with  $Y_{av}$ , dynamic lighting conditions can be accounted for. A sample image is below:

This approach worked extremely well, with us finishing equal first in this challenge. Although one problem was noticed at Robocup as our approach is better suited to a smooth change in light across the field. The direct lights used at Robocup meant the ball and the robot could be under vastly different lighting. Our approach changed the camera settings to values best suited to the location of the robot (not the location of the ball), this lead to distance balls not always being seen.



Figure 59: Shutter speed set to fast, lighting decreases  $Y_{av}=22$  therefore change shutter speed to medium.

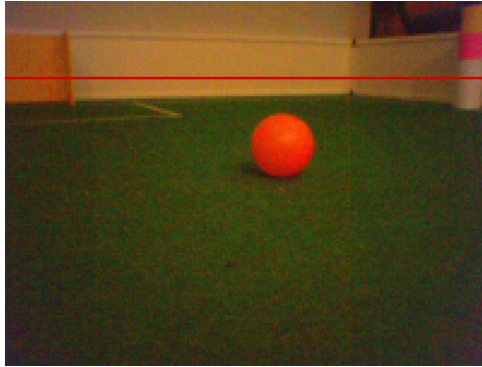


Figure 60: Result of image when shutter speed changed to medium speed.

### 9.3 Almost SLAM

The Almost Slam Challenge requires a large amount of collaboration between both the vision module and the localisation module. The vision module must be able to recognise the new objects that are present and re-recognise them as the same object. While the localisation module must keep track of the locations of the objects and then re-use them as the basis for localisation.

The solution developed for this challenge involved a form of coloured pattern recognition. The method used creates a selection of regions of the image centred on the pink blobs, taking advantage of the guaranteed minimum of three objects containing pink. Nine regions in total are created, equal to the size of the pink blob



to form a 3x3 cross. Each section of the grid is scanned for the two most common colours in that region. Once these have been found the colours of each region of the grid is compared to the equivalent region of each of the available patterns for each object that has been previously seen. If a similar pattern with eight or more sections of the grid having at least the most common colour, then the object is identified. During the initial exploration stage of the challenge the original beacons are used for localisation and so the location of the objects that are seen can be worked out as the robot has an accurate estimation of its current position on the field. In this stage if a pattern is not matched to an object the calculated location of the pattern is then compared with that of the previously seen objects. If it is within an acceptable range to the previously seen object it is assumed to be that object and the patterns are added as an additional pattern for that object. Otherwise the pattern is set as a new object. At the end of this phase the most commonly seen patterns are set as the land marks. During the localisation stage of the challenge objects are only identified, not created. This means that if a new pattern is found it is ignored. This occurs because the location of the object cannot be accurately calculated as it cannot be guaranteed that the robot can accurately predict its location as was the case in the previous section. For the module to calculate the distance, bearing and elevation to the objects in the second part of the challenge, the size of the blobs is tracked by the localisation module. During the initial part of the challenge the localisation module computes a linear relationship between the size of the blobs and the distance of the objects. This is then used by the object recognition code when finding objects in the second part of the challenge.

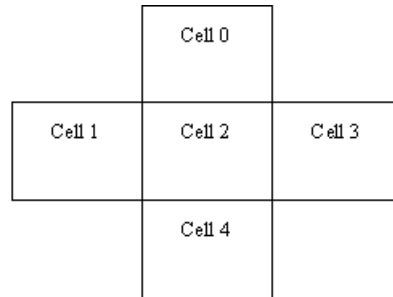


Figure 61: Scanning grid for almost SLAM Challenge

The results obtained offline using streams of test images featuring the normal

beacon objects showed promising results with four different objects repeatedly recognised a large number of times, these being the four distinctively coloured beacons. The system was also tested on the field with the beacon recognition code disabled. Because the system must work closely with the localisation module a large number of changes were also made to the localisation portion of the code by other members of the NUbots team. Unfortunately during The Almost SLAM Challenge event one of these changes caused a math error causing the robot to crash, and hence failing the event.

### **Challenge Results:**

1. Cerberus (Turkey)
2. NUbots (Australia)
3. rUNSWift (Australia)

## **Acknowledgements**

The NUbots are grateful to all colleagues, friends, previous members, and other supporters of the team including Dianne Piefke, Nick Hawryluk and other technical support and administrative staff in the School of Electrical Engineering & Computer Science and the Faculty of Engineering & Built Environment. Financial support for the NUbots' participation in RoboCup 2005 was provided by the ARC Centre for Complex Dynamic Systems and Control (CDSC), BHP Billiton Innovation and a Research Infrastructure Block Grant 2005 from the University of Newcastle in Australia.

Links to the NUbots' publications can be found at the NUbots' webpage

<http://robots.newcastle.edu.au/>

## **References**

- [Bunting *et al.*, 2003] J. Bunting, S. Chalup, M. Freeston, W. McMahan, R. Middleton, C. Murch, M. Quinlan, C. Seysener, and G. Shanks. Return of the NUbots ! The 2003 NUbots Team Report. Eecs tech report, University of Newcastle, 2003.

- [Chalup *et al.*, 2002] S. Chalup, N. Creek, L. Freeston, N. Lovell, J. Marshall, R. Middleton, C. Murch, M. Quinlan, G. Shanks, C. Stanton, and M.-A. Williams. When NUbots Attack ! The 2002 NUbots Team Report. Eecs tech report, University of Newcastle, 2002.
- [Henderson, 2005] N Henderson. Digital Image Processing In Robot Vision. Technical report, University of Newcastle, 2005.
- [Hengst *et al.*, 2002] B. Hengst, S.B. Pham, D. Ibbotson, and C. Sammut. Omnidirectional Locomotion for Quadruped Robots. *RoboCup 2001: Robot Soccer World Cup V*, pages 368–373, 2002.
- [Hong, 2005] K Hong. NUbots: Enhancements to Vision Processing, and Debugging Software for Robocup Soccer. Technical report, University of Newcastle, 2005.
- [Nicklin, 2005] S Nicklin. Object Recognition in Robotic Soccer. Technical report, University of Newcastle, 2005.
- [Quinlan *et al.*, 2004] M. Quinlan, C. Murch, T. Moore, R. Middleton, Li. Lee, R. King, and S. Chalup. The 2004 NUbots Team Report. Eecs tech report, University of Newcastle, 2004.
- [Röfer *et al.*, 2004] T. Röfer, T. Laue, H-D. Burkhard, J. Hoffmann, M. Jungel, D. Gohring, M. Lotzsch, U. Duffert, M. Spranger, . B. Altmeyer, V. Goetzke, O. v. Stryk, R. Brunn, M. Dassler, M. Kunz, M. Risler, M. Stelzer, D. Thomas, S. Uhrig, U. Schwiegelshohn, I. Dahm, M. Hebbel, Nistico W, C. Schumann, and M.Wacher. GermanTeam 2004. Technical report, 2004.
- [Seysener *et al.*, 2004] C. J. Seysener, C. L. Murch, and R. H. Middleton. Extensions to Object Recognition in the Four-Legged League. In D. Nardi, M. Riedmiller, and C. Sammut, editors, *Proceedings of the RoboCup 2004 Symposium*, LNCS. Springer, 2004.
- [Seysener, 2003] C Seysener. Vision Processing for RoboCup 2003. Technical report, University of Newcastle, 2003.