

THE UNIVERSITY OF NEW SOUTH WALES
SCHOOL OF COMPUTER SCIENCE & ENGINEERING

Real-Time Shared Obstacle Probability Grid
Mapping and Avoidance for Mobile Swarms

Joshua Shammai (3019911)
Bachelor of Science, Honours

Supervisor: William Uther
Assessor: Claude Sammut
Submitted: September, 2005

Abstract

In swarms of virtually identical robots the same programming code is often run on several different robots at the same time. This report shall expand upon a method for different instances of the same program to account for the subtle differences, specifically differences in vision systems, between the robots of the swarm.

Obstacle avoidance for mobile robots in a swarm can be a complex issue and very different than obstacle avoidance for mobile robots acting alone. This report shall also explore a method of mapping and avoiding moving obstacles in a known environment that takes into account the observations of other robots in the swarm.

Both these issues are explored as they apply to the rUNSWift team participating in the RoboCupSoccer Four Legged League in 2005.

Acknowledgments

Working on the rUNSWift team this year has been a great deal of work and a great deal of fun. I would like to thank Wei Ming Chen, Derek Leung, Nobuyuki Morioka, Alex North, Phu Quang, Victor Phung and Andrew Sianty as fellow team members and volunteer coders for helping to make this experience a thoroughly positive one.

I would also like to thank William Uther and Claude Sammut for their many helpful suggestions and analysis during the course of our programming. We would have gotten stuck a lot more if not for their knowledgeable advice.

A special thanks to Brad Hall for taking care of the organising side of things, there is no doubt that the entire team would currently be wandering lost around the streets of Osaka if not for him. Plus the team t-shirts would not have been nearly as cool.

Finally an extra thank you to William Uther for being the main driving force behind the rUNSWift team in 2005, we couldn't have done it without you Will.

Table of Contents

Chapter 1	Introduction	5
1.1	The International RoboCup Competition	5
1.2	The Four-Legged League	6
1.2.1	Overview	6
1.2.2	The Sony AIBO ERS-7 Entertainment Robot Dog	7
1.2.3	Environment and Rules	8
1.2.4	Changes to Environment and Rules Since 2004	10
1.3	The 2005 rUNSWift Architecture	13
1.4	Overview of This Thesis	15
Chapter 2	Linear Shifting	17
2.1	Overview	17
2.2	Calculating Linear Shifts	18
2.3	Reconfiguring Linear Shifts	24
2.4	Evaluation	27
Chapter 3	Previous Work on Obstacle Mapping and Avoidance	29
3.1	Related Work	29
3.2	Previous rUNSWift Strategies for Obstacle Avoidance	31
3.2.1	2003 Obstacle Avoidance Challenge	31
3.2.2	Gap Finding	32
3.2.3	Stealth Dog	33

Chapter 4	Obstacle Mapping	34
4.1	Obtaining Obstacle Points from Vision	34
4.2	Obstacle Probability Grid Mapping	36
4.3	Obstacle Box Sharing	41
4.4	Problems and Limitations	41
4.5	Possible Solutions	42
Chapter 5	Obstacle Avoidance	43
5.1	Path Planning	43
5.2	Best Gap	45
5.3	High-Level Strategy – Dodgy Dog	49
5.4	Problems and Limitations	52
5.5	Possible Solutions	54
Chapter 6	Conclusion	56
	Bibliography	58
	Appendices	61

Chapter 1:

Introduction

1.1 The International RoboCup Competition

RoboCup is an international competition and symposium held annually whose goal is to advance research in the fields of artificial intelligence and robotics. RoboCup consists of three main subdivisions; RoboCupSoccer, RoboCupRescue and RoboCupJunior.

RoboCupSoccer pits robots and artificial intelligence software against each other in the game of soccer, consisting of five separate leagues it is the largest subdivision and the biggest attraction of the RoboCup event. The leagues that make up RoboCupSoccer are as follows;

Simulation League: This league is unique among the soccer leagues in that it does not use any actual robots. Instead, teams of eleven autonomous software agents from different programming teams compete against each other in a simulated field.

Small Size Robot League: The soccer playing robots in the small size league must fit within a 180mm diameter circle and be no higher than 15cm. Each team is comprised of five such robots who play on a field 2.8m by 2.3m. An orange golf ball is used as the soccer ball.

Middle Size Robot League: The middle size robot league consists of teams of a minimum of two robots and a maximum six robots each, the total space occupied by all players of the team may not exceed 10,000cm². Middle sized robots play soccer with an orange FIFA standard size 5 football.

Four-Legged Robot League: The four legged league consists of soccer games between two teams of four Sony AIBO robot dogs. For more information on this league see section 1.2.

Humanoid League: The humanoid league is definitely one of the most challenging in RoboCupSoccer, especially given that the field of humanoid robotics is far from mature. However it is also one of the most necessary leagues towards RoboCups' ultimate goal of beating the world soccer champions by the year 2050.

RoboCupRescue tests the ability of robots and artificial intelligence software to rescue simulated victims in simulated disaster situations and locations. Although RoboCupSoccer is lots of fun and a great way to engage interest and advance the field, RoboCupRescue advances hardware and programming techniques that have far more direct and important uses in the real world. RoboCupRescue consists of two leagues as follows;

Rescue Simulation League: The simulation league creates a complex simulated disaster environment with software agents representing real-world disaster workers such as firefighters, victims, commanders, volunteers, etc. The enormity of the research is often breathtaking as agents must use advance planning and initiative to bring the disaster under control.

Rescue Robot League: In the rescue robot league teams send in multiple autonomous or tele-operated robots to simulated disaster arenas of varying difficulty/complexity levels. It is the robots job, through the tele-operator if necessary, to seek out and identify victims and place them within a map it builds of the arena as it goes along.

RoboCupJunior is a subdivision aimed squarely at the younger generation in an effort to engage the interest of children in the field of robotics and artificial intelligence and to hopefully ensure RoboCup has participants well into the future. RoboCupJunior consists of three main parts: Soccer, Rescue and Dance.

1.2 The Four-Legged League

1.2.1 Overview

The RoboCupSoccer Four-Legged League is one of the most interesting in the entire event. Through the standardization of the robot players in the league the competition is able to focus solely on the programming of the players. The standardised robot that is used is the Sony AIBO entertainment robot dog, this comes in a variety of model numbers and this years rules allow for the use of the following four models (ref. [2])

- Sony AIBO ERS-210 (black (i.e. dark gray) or white)
- Sony AIBO ERS-210A1 (black (i.e. dark gray) or white)
- Sony AIBO ERS-7 (white)
- Sony AIBO ERS-7M22 (white)

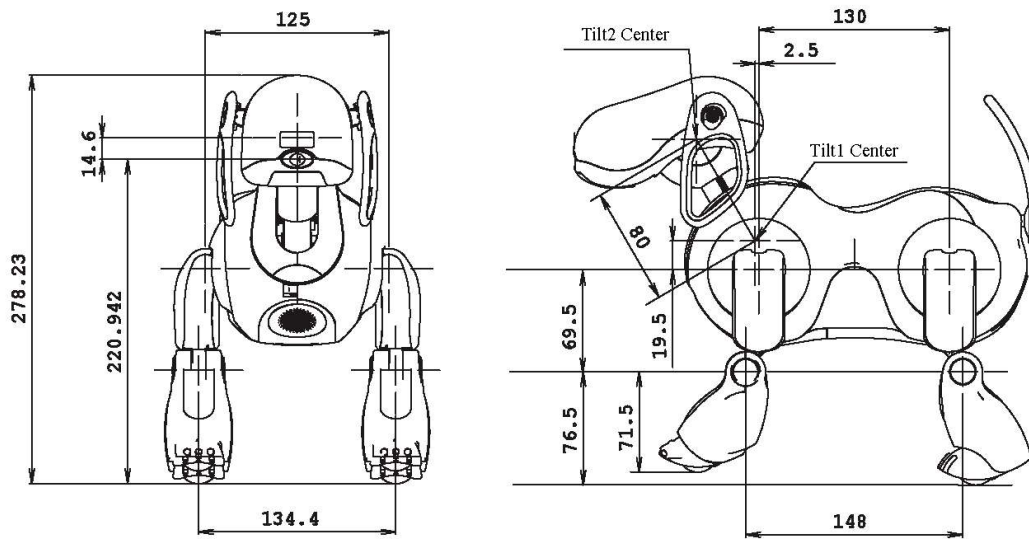
This year the rUNSWift team competed with brand new Sony AIBO ERS-7 models.

1.2.2 The Sony AIBO ERS-7 Entertainment Robot Dog



The Sony AIBO ERS-7 Entertainment Robot has a 64-bit RISC processor running at 576MHz and 64MB of memory. As well as a 300,000 pixel CMOS image sensor with a 640 by 480 image output able to take pictures at 30 frames a second. The AIBOs lithium ion battery will last for about one and a half hours under normal use however RoboCup is far from normal use, the dogs are pushed far past their initially intended operating parameters and the batteries tend to last about 20 minutes for a dog that is continually playing soccer, or the length of a single match.

Programming for the AIBO is done via an API maintained by Sony known as OPEN-R. Programs written with OPEN-R in C++ are compiled onto special Sony AIBO memory sticks, which can be inserted into an onboard memory stick reader on the underbelly of the AIBO, the dog will then load with the OPEN-R programming rather than its default behaviour set. The side and front schematics of the AIBO are shown in Figure 1.



unit: mm

Figure 1: AIBO Schematics (ref. [4])

1.2.3 Environment and Rules

The Sony AIBO dogs play a game of soccer on a green field 400cm long by 600cm wide. The field has a goal at either end, one blue and one yellow, each 30cm wide and 80cm deep. Four beacons surround the field, each with a unique pattern of either blue and pink or yellow and pink. The beacons and goals are uniquely coloured in order to assist the dogs in localising themselves on the field. The dogs play in sticky backed velvet uniform pieces, the team playing in red uniforms defends the yellow goal and the team playing in blue uniforms defends the blue goal. At half time the dogs switch uniforms and sides in order to ensure any possible disadvantage caused by uniform colouring, such as seeing the ball in the red uniform, is distributed evenly.

The game is played in two halves of ten minutes each, half time consists of a ten minute break during which teams change the uniform for the opposite colour, and may change batteries, dogs, or memory sticks. It is not unknown for a team to fix a coding bug during half time and recompile the teams memory sticks, so long as they are quick about it. Each team may call one five minute timeout per game to be used at their discretion, however the majority of games occur without either side needing to use it.

The robot players are completely autonomous, all decision making during a game is done on board without the intervention of the programmers. There is a wireless network covering the field which the dogs may use to communicate with their team members. The dogs should also be programmed to respond to the GameController program which communicates via the wireless network. The GameController program acts as a referee, assigning penalties, announcing changes in game states, keeping official game time and telling players when the ball has gone out of the field and who kicked it out since this will determine where it is placed back on the field.

During play there is a head referee and two assistant referees on the field. The referees make official decisions regarding goals, penalties, dog placement in ready state, etc. In the case of disputed decisions the head referee has the final say.

Penalties generally incur the price of being taken off the field for thirty seconds, two penalties especially relevant to the material in this thesis are

Player Pushing: Pushing another player for three seconds, excepting the player closest to the ball or a robot standing still, is a penalty incurring thirty seconds out of play. Obstacle avoidance is a lot more necessary because of this rule.

Leaving Field: A dog leaving the field, excepting one that is making an effort to get back providing it isn't taking too long, is given a thirty second penalty. This is the first year this penalty is possible (without some extremely tricky moves to get over the wall) but it is important to both linear shifting and obstacle avoidance. Linear shifting because it helps with vision and thus with the robot knowing where it is on the field and obstacle avoidance because the dog must be sure to try not to dodge obstacles in such a way that it ends up off the field.

There are several playing states that a robot may be in, determined by messages from the GameController or by specific combinations of buttons pressed on the robot should the wireless network be down.

Initial: Robot should not move in any fashion except standing up.

Ready: In this state the robot should localise itself on the field and move to its correct starting position.

Playing: In this state robots are playing soccer.

Penalised: This is the state robots go into when they have incurred a penalty, the robot should not move.

Finished: This is the state reached when the half or game has ended, robots need not move in this state.

1.2.4 Changes to Environment and Rules Since 2004

Extensive changes have been made to the playing environment since last year. Firstly and most importantly, as show in Table 1 the playing field has increased in size by over 70%, this in itself is enough to enable new strategies and playing techniques. The extra field space is excellent in terms of obstacle avoidance as there is a lot more space to use while dodging, however if the obstacle avoidance is not tuned well then the extra space can translate into a longer path and a slower dog. With this years larger field vision and object detection must be especially sensitive as well, an object across the field in 2005 may be 150cm further away than an object across the field in 2004 and teams that can deal well with this will gain an advantage in play. Field diagrams for last year and this year are provided in Figure 2 and Figure 3.

The second major change is that for the first time in the league there are no walls surrounding the inner playing field. This effectively means there is nothing to stop the ball or the robots wandering outside the designated playing area apart from their own programming. A dog must be very careful when avoiding obstacles as to not avoid them right off the field. In terms of vision there is now a greater problem of discarding things seen outside the field instead of trying to classify them as objects. This has always been a problem in RoboCup however it is more pronounced without the buffer the walls provided.

Other changes, important though not as significant as the first two, include the repositioning of the beacons from the corners of the field to positions on the side of the field at the center of each field half. As well, the goals have been made wider but less deep and it is worth noting that it is now possible for a robot to get beside the goal on a level with the goal wall, this was not possible with the field walls last year and presents a challenge for the goalie more than other players as it tries to maintain position inside the goal box. If it drifts away it can get stuck against the goal wall as it tries to get back into position. Having the goalie navigate around the goal walls is a necessity when it is returning to its home position.

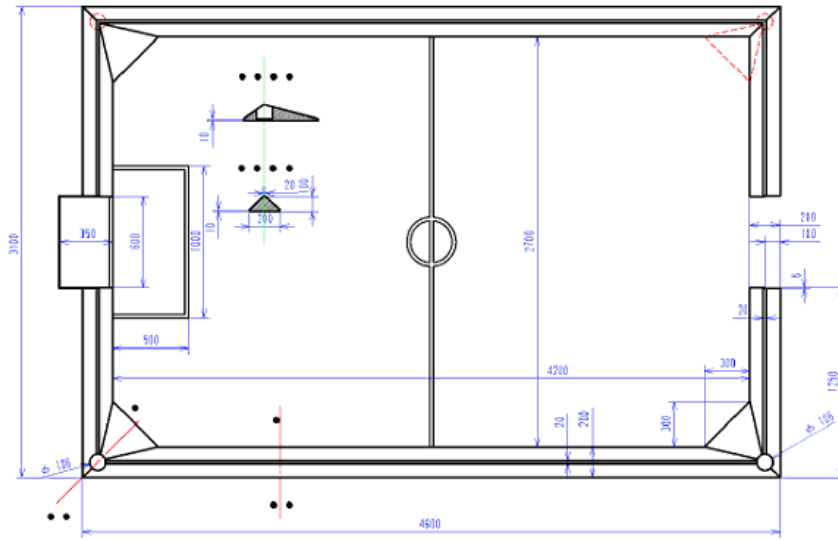


Figure 3: Field Dimensions from 2004 (ref. [3])

	Playing Field	Goals
2004		
Dimensions (mm)	2,700 by 4,200	350 by 600
Area (mm ²)	11340000	210000
2005		
Dimensions (mm)	3,600 by 5,400	300 by 800
Area (mm ²)	19440000	240000
2004 to 2005		
Area Increase (mm)	8100000	30000
Area % Increase	71.43%	14.29%

Table 1: Comparison of Field Dimension from 2004 (ref. [3]) to 2005 (ref. [2])

There have been few significant changes to the rules this year, mostly minor modifications relating to the changes to the environment, such as adding a penalty for leaving the field which was not possible in previous years.

1.3 The 2005 rUNSWift Architecture

The rUNSWift team programs the Sony AIBOs using the OPEN-R API (ref. [4]), this is a free Software Development Kit (SDK) provided by Sony to allow users of the AIBO to interact with it at the lowest levels. OPEN-R is based on a modularised software concept with many interconnected program modules or 'objects' able to run concurrently and communicate with each other.

Though low level programming is done in C++ to be compatible with the OPEN-R framework, high level strategies are programmed in Python. The Python code base has ties into the C++ code base, meaning that the Python code may communicate and interact with the C++ code and vice versa. Because Python is a higher level language than C++ its compiled code can sometimes be a bit slower, for this reason all attempts are made to program any processor intensive algorithms in C++.

The 2005 rUNSWift code base can be broken up into several interacting modules;

GPS Module: The GPS module keeps track of objects with known locations, this includes other teammates and the ball, but not opponent dogs. Most importantly this module uses beacon and goal location, odometry and line detection to keep track of the dogs current location and heading. In RoboCup terms the dogs ability to keep track of its own location on the field is known as localisation, the GPS module also keeps a certainty value relating to the probable accuracy of current localisation position and heading.

Obstacle Module: The obstacle module is the one that will be discussed most thoroughly in this report, this module is responsible for keeping track of where all the obstacles are and performing operations over the obstacle grid. Really the obstacle module is part of the GPS module however for the purpose of this report it is easier to think of it as a module in its own right.

Vision Module: Every frame the vision module takes the image from the camera and processes it, extracting useful information such as the location of any beacons, goals, obstacles or balls in sight. Gathered information is passed to the GPS module, except for obstacle points which are passed to the obstacle module. The vision module will map points from an image into x,y,z coordinates before passing them on.

The rUNSWift vision module uses the YUV colour model when working with images taken from the camera. The YUV model is similar to RGB however, where each element of RGB is a colour component, only U and V are colour components in YUV and Y is a measure of brightness or luminance. There is a direct mapping between RGB and YUV as shown in Figure 4.

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Figure 4: RGB to YUV conversion matrix (ref. [15] p. 40)

Behaviour Module: Behaviour is the top level module responsible for decision making and game play. This module is mainly implemented in python, which has a slower runtime compared to C++ however is still a far easier tool to use for behaviour programming from the programmers point of view because of the intuitive programming syntax structure.

Communication Module: The communications module is responsible for communication over the wireless network with other dogs and with the game controller. This module is merely a message passer and no actual processing of messages is done here, a received message is sent to other modules to deal with and messages going out are sent to the communications module already fully formed.

Actuator Control Module: The actuator control module occupies the lowest level on the chain providing the interface between the C++ code and the physical hardware components. Preprogrammed movements such as kicks and blocks can be defined in POS files, these are files that contain sequences of joint positions in a format that the Actuator Control Module can read.

All decision making processes must be carried out in just under 30 milliseconds (ms), each such processing cycle is referred to as a 'frame'. This is a time set by the speed at which images come in through the onboard camera, a processing cycle that takes longer than this results in 'dropped frames'. Dropped frames refers to instances where the robot skips one or more processing cycles because a previous cycle has gone overtime. Needless to say these are very undesirable and can result in stuttering of the robots' walk and deficiencies in the data making up its current world model. Note that the rUNSWift architecture pushes right up against the 30ms limit as a lot of things must occur within this time, image processing, behaviour decisions and various data upkeeping tasks to name a few. Any new algorithm that takes more than one or two milliseconds without saving an equal or greater amount of time cannot be added without causing frame dropping.

1.4 Overview of This Thesis

This thesis attempts to give details about some of the developments made by the 2005 rUNSWift team in the fields of linear shifting and obstacle avoidance. This thesis on its own is by no means a full guide to the work done by the 2005 rUNSWift team and for further information interested readers are referred to the thesis reports of the other team members (ref. [6,7,8,9]).

The two main questions this thesis will be addressing are;

How do the robots deal with obstacles during play?

In answer to which we will show the robots have a sophisticated internal mapping of the constantly moving obstacles on the field which they use to derive the heading of the best path to follow on the course to their chosen destination.

And

How are the differences between the individual dogs accounted for when programming? Here our answer will focus on accounting for differences between the dogs' onboard cameras by providing details on the methods used to shift incoming camera images to appear as if they had been taken by another dog. This allows colour calibration in new environments to be done for a single dog only. Camera images are not the only issue in individual dog characteristics however it is the only one that will be covered in this report.

Chapter 1 provides an introduction to RoboCup, the Four-Legged League, the Sony AIBO ERS-7 and the rUNSWift code base.

Chapter 2 covers the topic of Linear Shifting, which refers to the calculation of a lookup table to shift the pixels of images so that they appear to have come from a different camera. Much of the work involved in calculating the master table has been automated this year as well as the addition of a program to reconfigure the linear shift table.

Chapter 3 provides a background on previous work done in the fields of obstacle mapping and avoidance, both by rUNSWift and by others.

Chapter 4 goes into the details of how Obstacle Mapping was tackled this year. A new Obstacle Probability Grid system is in place providing robots with locations of clouds of obstacles derived through shared observations of the environment.

Chapter 5 will explain how a new skill has been added to the rUNSWift dogs to handle Obstacle Avoidance, that of Dodgy Dog, utilising the obstacle clouds to pick best gaps to head for and using sequences of best gap paths with intelligent strategy decisions to reach destinations on the field while avoiding obstacles.

Chapter 6 concludes the thesis giving an overview of the progress achieved in Linear Shifting, Obstacle Mapping and Obstacle Avoidance.

Chapter 2:

Linear Shifting

2.1 Overview

The inbuilt cameras on the Sony AIBO dogs, like cameras everywhere, are extremely sensitive to subtle changes in the way they are built with the result that no two cameras are exactly alike. Even taking a photo from the exact same location at the exact same time with two different but supposedly identical cameras, if such a feat were possible, would still result in different images being captured. For the large part this is not noticed and irrelevant for usage in mainstream photo taking applications because the changes are very subtle and not noticeable to the human eye. However they start to become a problem when a computer is relying on an exacting subset of YUV values to determine the colour of an object.

The average user would be surprised how much even a slight change in lighting conditions can greatly effect how the robots classify, or misclassify, objects on the field. For every different field, or for every time a field is moved or has its lighting conditions changed, a set of calibration images must be taken and from these various tools are used to generate a colour classification table. For more information on this process and the sensitivities and limitations of the vision system the reader is directed to the rUNSWift report of Alex North (ref. [6]).

In theory because each robots camera is different this process of recalibration for each new field should be done for every single robot. However, even calibrating for a single robot takes a prohibitive amount of time, doing the same for four or eight is not a pleasant prospect. Usually the differences in the camera are small enough to just calibrate for one robot, the base dog, and accept the small amount of misclassification the non-base dogs will experience. Linear shifting is a procedure used to minimise this error by calculating transformation formulas for each dog to transform images from their camera and make it look as if it was taken from the base dog.

Linear shifting is not new to this years rUNSWift team, last years team used a linear shift table as well which was introduced by team member Jing Xu (ref. [12]). The work done on linear shifting this year has been in largely automating the process of calculating the master linear shift table, so that all that is needed now is a correctly labeled set of images in their own subdirectory and the linear shift program will do the rest. As well, a program has been developed to reconfigure a linear shift table for a new base dog, the details of which will be discussed further in section 2.3.

2.2 Calculating Linear Shifts

Every image taken by the dogs camera is stored as an array of pixels, and each pixel is comprised of three values; Y, U and V. To calibrate a linear shift for a non-base dog three transformation formulas are found, one each for the three values of Y, U and V. These formulas can then be run over each pixel in the image array and the image is thus transformed to look like it was taken by the camera of the base dogs. Note that the base dog is the dog for which the colour calibration has been performed.

Each formula has the form $y = m \cdot x + b$ where y is the transformed value and x is the raw value of whichever of the three components the formula is for, the three formulas are shown in Figure 5. Calculating a linear shift for a dog therefore consists of calculating an 'm' and 'b' for each of the three image components Y, U and V. The premise for calculating these values is relatively simple, though the implementation of the calculation can be time consuming. Photos are taken of single colour sheets under the same field and lighting conditions by different dogs. The YUV values of these images are analysed and plotted on a graph, after which an estimation of the line of best fit is calculated.

$$y_{new} = m_y \cdot y_{old} + b_y$$

$$u_{new} = m_u \cdot u_{old} + b_u$$

$$v_{new} = m_v \cdot v_{old} + b_v$$

Figure 5: transformation formulas for linear shifting

For this procedure several matte colour sheets were used, each one with an unvarying colour across its surface and at least the size of an A4 sheet of paper. The colours used for the rUNSWift linear shift calibration were RoboCup beacon pink, RoboCup dog-uniform red, RoboCup dog-uniform blue, RoboCup beacon blue, RoboCup 2003 beacon green, RoboCup beacon yellow, and white. For each colour a series of at least ten photos was taken with the base dog and with the dogs that were having their linear shifts calculated. To make sure the environment was as consistent as possible between different dogs the position of the first dog and the first sheet was marked with tape so that subsequent dogs and sheets could be aligned to the same positions. As well, photos were taken as close together in time as possible to minimise light changes from natural lighting, artificial lights were kept at the same level.

Due to the overhead time involved in setting up it is preferable to get photo series for as many dogs as possible at once, in situations when this was not be possible, for every time a new dog was calibrated another series of photos was taken with the base dog for that particular occasion and when calculating the transformation for a non-base dog the images from the base dog that were taken at the same time were used.

The key algorithm in calculating linear shift is the least squares fitting method of linear regression (see ref. [24] for a good background on this). Least squares fitting deals with taking a set of (x, y) points and calculating the formula of the best fitting line. Where a line formula is taken to be $y = m \cdot x + b$, least squares fitting will calculate the constants 'm' and 'b'. Least squares fitting is best thought of not as a way of finding values in an equation but of minimizing the sum squared error in an equation, which is how the method gets its name. To fully explain this mathematical method an example situation from linear shifting shall be worked through.

To calculate the shift equation for a dog, A, with base dog, B, just for the U component of YUV a series of photos is taken of our colour sheets with both dogs under identical conditions.

Taking red as an example colour; for each red photo taken by the base dog the U value is analysed and an average U value for all the red photos is found, let this value be

$$U_{Red}^B \quad (2.2.1)$$

This shall be used as the absolute value of U for the colour red as seen by the base dog. Now each red photo taken by dog A is analysed and a set of U values is extracted with several values taken from each photo;

$$U_{Red1}^A, U_{Red2}^A, U_{Red3}^A, \dots, U_{RedN}^A \quad (2.2.2)$$

This set represents the colour red as seen by dog A.

Value (2.2.1) and set (2.2.2) are merged into a set of (x,y) coordinates like so;

$$\{(U_{Red}^B, U_{Red1}^A), (U_{Red}^B, U_{Red2}^A), \dots, (U_{Red}^B, U_{RedN}^A)\} \quad (2.2.3)$$

Significantly it is the base dog value that is used for x when logic tells us that it should actually be y if the final formula is meant to translate a value from the vision system of dog A to the vision system of base dog B. The reason for this is that least squares fitting minimises the error *along the y axis*. It does this by minimizing the squared values of the vertical offsets between the x values and the line as shown in Figure 6. Because x and y are being used the wrong way around it is important to remember after calculating the values that the function that was calculated is actually the inverse of the function that is needed, though it is an easy matter to remedy as will be shown at the end of these calculations.

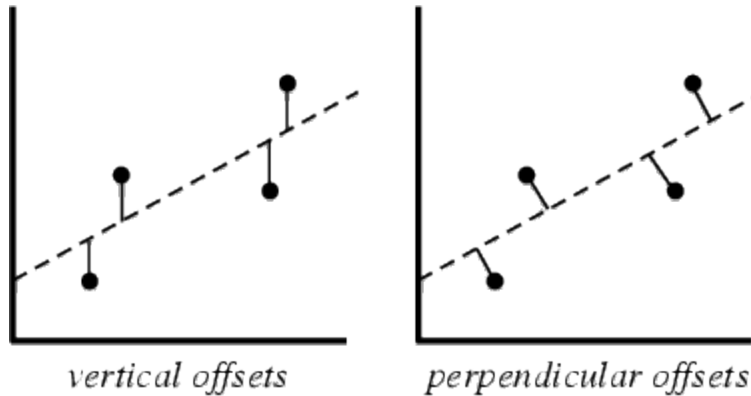


Figure 6: Vertical offsets, not perpendicular, are used in calculating the line of best fit (image taken from ref. [16])

Note that it might seem to make more sense just to use the vertical offsets instead of their squares however it turns out to be far easier mathematically to use the squared values, though this comes with the trade off that outlying points are given undue weighting.

If set (3) is plotted graphically it will look something like the graph in Figure 7

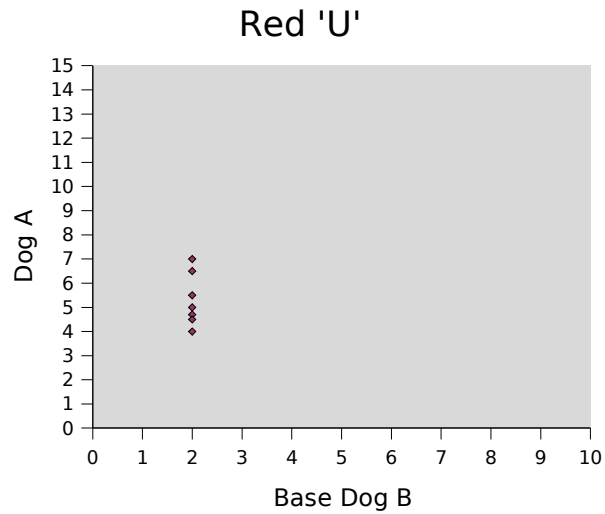


Figure 7: Example value for U derived from colour red on Base Dog B and Dog A

Now the same data gathering is applied to the rest of the colours and seven sets of data points are acquired;

$$\begin{aligned}
& \left\{ (U_{Red}^B, U_{Red1}^A), (U_{Red}^B, U_{Red2}^A), \dots, (U_{Red}^B, U_{RedN}^A) \right\} \\
& \left\{ (U_{Blue}^B, U_{Blue1}^A), (U_{Blue}^B, U_{Blue2}^A), \dots, (U_{Blue}^B, U_{BlueN}^A) \right\} \\
& \left\{ (U_{LBlue}^B, U_{LBlue1}^A), (U_{LBlue}^B, U_{LBlue2}^A), \dots, (U_{LBlue}^B, U_{LBlueN}^A) \right\} \\
& \left\{ (U_{Pink}^B, U_{Pink1}^A), (U_{Pink}^B, U_{Pink2}^A), \dots, (U_{Pink}^B, U_{PinkN}^A) \right\} \\
& \left\{ (U_{Yellow}^B, U_{Yellow1}^A), (U_{Yellow}^B, U_{Yellow2}^A), \dots, (U_{Yellow}^B, U_{YellowN}^A) \right\} \\
& \left\{ (U_{Green}^B, U_{Green1}^A), (U_{Green}^B, U_{Green2}^A), \dots, (U_{Green}^B, U_{GreenN}^A) \right\} \\
& \left\{ (U_{White}^B, U_{White1}^A), (U_{White}^B, U_{White2}^A), \dots, (U_{White}^B, U_{WhiteN}^A) \right\}
\end{aligned}$$

The graph for all of these will look something like the graph in Figure 8

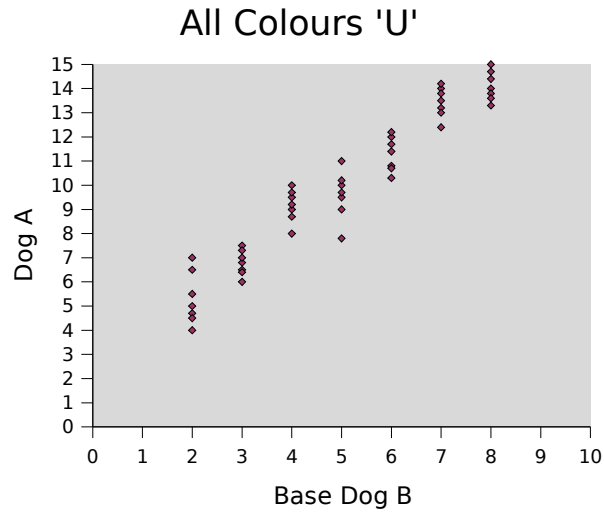


Figure 8: Example value for U derived from all seven calibration colours on Base Dog B and Dog A

To get the line formula the least squares fitting algorithm is applied to the data, this calculates the constants 'm' and 'b' such that the sum squared error over the line, written mathematically as $\sum (y_i - f(x_i))^2$, is minimised. Note that the error cannot be gotten rid of all together as that would mean the end equation would have to trace a line in Figure 8 that touched every point in the graph. The minimised error is given in the output of this years new linear shift calculating program. It is not necessary to go into the details of how these values are actually calculated, there is a wealth of material in the mathematics world and on the internet for anyone who is interested, it is just necessary to understand how to prepare the data for the algorithm and how to treat the values output from it. Least squares fitting was actually performed by entering the relevant data into a spreadsheet program and running the LINEST function, any decent spreadsheet or mathematics program will have a least squares fitting algorithm available to use.

At this point it must be remembered that the output data is actually the inverse of the sought after function. The formula calculated, $y = m \cdot x + b$, is valid where y is the base dogs' U value and x is the U value for dog A, for linear shifting we actually need to translate images from dog A to the base dog B, not the other way around, so the formula must be inverted.

$$y = m \cdot x + b$$

$$y - b = m \cdot x$$

$$x = \frac{(y - b)}{m}$$

$$x = \frac{1}{m} \cdot y - \frac{b}{m}$$

$$m_{inverted} = \frac{1}{m}$$

$$b_{inverted} = \frac{-b}{m}$$

The inversion gives the final formula values, although these are only the values for the U value of a single dog, this process has to be repeated for the Y and V values of dog A and the Y, U and V values of all other dogs for which a linear shift is required. The linear shift program calculation program developed this year will read over a set of labeled images and produce a spreadsheet file for each non-base dog containing its linear shift.

Note that calculating linear shift is no small task, rUNSWift needed at least eight dogs to practice full games, each dog required ten photos for seven colours so that for eight dogs five hundred and sixty images needed to be taken and analysed. Even with the new program to automatically generate the linear shift table there is still a lot of work that needs to be done. Though calibrating eight dogs for a new environment would still have taken far more time than calibrating one dog and a linear shift table.

2.3 Reconfiguring Linear Shifts

It is not unknown that through the extreme rigors of RoboCup matches a dog may suffer a serious fault, in fact as the teams force the dogs to do more and more every year it is regrettably becoming an increasing problem. Should the base dog become incapacitated the linear shift table in effect becomes useless for any new field because the colour calibration will have to be done on a new dog. We have already mentioned that calculating linear shift is a laborious process and it is entirely conceivable situations can arise where a new table is needed but there is no time to calculate one.

To overcome this problem a program was developed this year to take a linear shift table and reconfigure it for another base dog. Note that this method will only work for a new base dog that is currently a non-base dog with a linear shift formula already calculated in the master table. The reconfigure shift program is made possible by the following mathematical formulas.

We are looking for the linear shift formula to transform a value from a non-base dog to a new base dog;

$$y = m_{new} \cdot x + b_{new}$$

The following formulas are known;

The linear shift from the non-base dog to the current base dog;

$$y = m_{old} \cdot x + b_{old} \quad (2.3.1)$$

The linear shift from the new base dog to the current base dog;

$$y = m_{base} \cdot x + b_{base}$$

However we actually need the formula to shift the current base dog to the new base dog which is the inversion of this equation;

$$x = \frac{(y - b_{base})}{m_{base}}$$

For readability in the final formula we swap x and y in this formula;

$$y = \frac{(x - b_{base})}{m_{base}} \quad (2.3.2)$$

So the formula for the shift from the non-base dog to the new base dog is found by substituting x in formula (2.3.2) with the value for y in formula (2.3.1) to give;

$$y = \frac{(m_{old} \cdot x + b_{old} - b_{base})}{m_{base}}$$

$$y = \left(\frac{m_{old}}{m_{base}} \right) \cdot x + \frac{(b_{old} - b_{base})}{m_{base}}$$

Meaning the constants in the sought after formula are;

$$m_{new} = \frac{m_{old}}{m_{base}}$$

$$b_{new} = \frac{(b_{old} - b_{base})}{m_{base}}$$

The new m and b need to be calculated for each Y, U and V for every non-base dog in the current table to generate the new linear shift table. The linear shift for the new base dog should be removed from the table as the base dog does not need a linear shift, though it is a good idea to keep track of which dog is the base dog in a comment in the table code. The linear shift for the old base dog is simply the inversion of the old linear shift for the new base dog which is formula (2.3.2) in the above calculations, for which the constants are;

$$m_{inverted\ base} = \frac{1}{m_{base}}$$

$$b_{inverted\ base} = \frac{-b_{base}}{m_{base}}$$

Strictly speaking the linear shift for the old base dog is probably not needed in the reconfigured table as reconfiguring a table is meant to be a last resort for when the base dog fails, it is included in these calculations merely for completeness and because of the relative ease with which its formula is obtained.

2.4 Evaluation

Reconfiguring the linear shifts introduces a certain amount of error. This was evaluated experimentally by generating multiple linear shift tables from the same set of calibration photos, each with a different base dog, the first table generated was generated with the base dog X and shall be referred to as table A_X . The second table was generated with the base dog Y and shall be referred to as table B_Y . The reconfigure shifts program was run over table A_X to give a table with base dog Y referred to as A'_Y , the error was then measured between table B_Y and table A'_Y as shown in Table 2.

	Average Relative Error	Maximum Relative Error
Y m b	0.0121 0.153	0.0002 0.1664
U m b	0.0461 0.1196	0.1004 0.2425
V m b	0.0089 0.0338	0.0144 0.0594
ALL m b	0.0223 0.1021	0.1004 0.2425

Table 2: error from reconfiguring linear shift

Note that a larger error in a b value is acceptable and in many ways expected as m is the main modifier for the formula. The error in reconfiguring the linear shift table was found to be acceptable and though using photo sets to generate a linear shift table is definitely a preferable method reconfiguring is a viable option should the situation necessitate it.

Chapter 3:

Previous Work on Obstacle Mapping and Avoidance

3.1 Related Work

In 1985 researchers Moravec and Elfes at Carnegie-Mellon University proposed a Certainty Grid to map dynamic obstacles onto a model of the environment (ref. [17, 18]). A certainty grid as they envisioned it is a two dimensional grid that can be overlaid on the environment with each grid square containing a measurement of the certainty that an obstacle exists at the corresponding location in the environment. Note that in this report the certainty grid based mapping system that is used by rUNSWift shall be referred to as the obstacle probability grid, the detailed workings of which are explained in Chapter 4.

Around the same time Khatib (ref. [20]) and Krogh (ref. [21]) explored the 'artificial potential field' concept, this was the first work which took obstacle avoidance away from high level path planning methods with a lower level simplification. In the artificial potential field method an imagined repulsive field surrounds obstacles, pushing against the robot, while a corresponding attractive field surrounds the destination, pulling the robot towards it. This was not intended to replace path planning at the higher level but merely to supplement it with a method that lent itself to faster, real-time processing.

In 1989 Borenstein and Koren proposed the Virtual Force Field (VFF) method for obstacle avoidance (ref. [19]), combining certainty grids and artificial potential fields. The VFF method keeps a world model for obstacles in the form of a histogram grid where each cell represents the confidence that there is an obstacle there. The histogram grid differs from the certainty grid in that it only increments a single cell per sensor reading creating a probability distribution. Obstacles probabilities in the histogram grid exert a virtual repulsive force on the robot in proportion to their confidence level and inversely proportional to the distance from the robot. The target destination exerts an attractive force on the robot thus moving it along its path. Every processing cycle the robot will sum the vectors of the repulsive and attractive forces acting on it to derive a heading and speed.

Limitations of the VFF include 'doorway' type situations, so called because they involve two close obstacles with enough gap between them for the robot to pass through but with both obstacles causing a combined repulsive force strong enough so that a robot cannot enter.

A similar situation occurs when going through a corridor, the path of the robot can start zigzagging in an inefficient fashion very easily if it gets too close to one wall and the repulsive force pushes it too far the other way. Another limitation is that the distance component of the repulsive force will occasionally fluctuate wildly between frames, causing correspondingly large fluctuations in the derived heading, which are a problem when attempting to navigate a stable path. The fluctuations can be controlled with a low-pass filter however the delay this introduces reduces the effectiveness of the algorithm against sudden unexpected obstacles.

To combat these limitations in 1991 Borenstein and Koren advanced on this strategy with the Vector Field Histogram (VFH) method (ref. [22]). The key difference from VFF is a two-stage data reduction in deriving heading, the histogram grid method of storing obstacles remains the same. In VFH a new heading is derived each processing cycle, in order to derive a heading a local polar histogram is first created from the histogram grid. The curve of the polar histogram is smoothed to give a histogram with 'peaks', areas of high obstacle concentration, and 'valleys', areas of low obstacle concentration. VFH selects a valley from the candidate valleys that has a heading closest to the destination heading which it uses to choose a direction of travel.

The power of VFH to overcome some of the limitations of VFF is in its ability to distinguish between wide and narrow candidate valleys. A narrow valley is indicative of a doorway or corridor situation, and recognising this the algorithm is able to adjust its heading accordingly.

Borenstein and Ulrich further developed VFH to produce VFH+ in 1998 (ref. [23]). VFH+ introduces several improvements over VFH with the addition of a hysteresis model as well as heuristics that take into account both the robots specifications and its kinematic properties, adjusting derived headings for turning time and only approaching gaps a robot of the given size can actually navigate. VFH and VFH+ share more than a few similarities to the Best Gap algorithm and the overlaying Dodgy Dog skill developed this year, both of which are examined in Chapter 5.

3.2 Previous rUNSWift Strategies for Obstacle Avoidance

3.2.1 Obstacle Avoidance Challenge

In 2003 one of the technical challenges for the four legged league was the obstacle avoidance challenge. It involved placing 7 obstacle robots standing still at random positions on the field, the robot taking the challenge then had 3 minutes to navigate across the field with as little collisions as possible. Although this challenge was discontinued in 2004 this section shall nevertheless give a background on one of the strategies developed in 2003 for the challenge named Gap Finding. This strategy is relevant because of its resemblance to the Best Gap algorithm developed this year for obstacle avoidance in the game. Gap Finding was one of four strategies investigated by the 2003 rUNSWift team in regards to this challenge.

In chronological order the four strategies were;

- 1) Force Field
- 2) A* Search
- 3) Gap Finding
- 4) Reinforcement Learning

Each subsequent strategy attempted to address limitations of the previous strategy ending with the Reinforcement Learning, which was the strategy used in the competition. Full details on all four attempted strategies can be found in Eileen Maks' thesis report for 2003 (ref. [11]).

It is worth noting that comparing obstacle avoidance strategies from the challenge to the ones in the game is not entirely fair. A robot taking 3 minutes to get to the other side of the field avoiding obstacles during an actual match would be unbelievably slow. Plus there is the fact that the obstacles in the challenge do not move and if you are certain of your location you can plot them as permanent fixtures in your world model, a very different situation to a proper match. Finally, even the slightest touch to an obstacle robot was severely penalised in the challenge whereas a certain amount of bumping into each other is expected in the frantic rush to the ball of a real game, so long as it does not violate the conditions for a player pushing penalty. Despite these differences we still consider examining the previously explored Gap Finding obstacle avoidance strategy as a beneficial background to this years strategy.

3.2.2 Gap Finding

The internal obstacle world model of 2003 for the obstacle avoidance challenge was similar to this years. Obstacles were represented internally as a certainty grid of obstacle clouds which, although the implementation and algorithms used were very different than this years, may still for convenience be thought of as the same as this years obstacle grid mapping as described in Chapter 4. The way gaps were selected based on that world model is entirely different. To select a gap a valid range is defined, usually 60 degrees to the left and right of the heading to destination. Within the valid range the largest gap is selected, size is not the only criteria though, the algorithm to find the gap also contains various biases to head away from corners and not lead the robot into dense obstacle patches and things of that nature.

The Gap Finding strategy consisted of four actions performed in a continual loop;

- 1) *Face the target, initially the heading to the destination*
- 2) *Scan for obstacles, calculate the best gap from obstacles observed*
- 3) *Rotate to face the calculated best gap*
- 4) *Walk through the gap, go back to 1)*

This strategy was ultimately rejected for use in the obstacle avoidance challenge for several reasons;

- *Reliance on obstacle distances*: The vision component of this strategy suffered from noise in distance measurements to the obstacles. Though instinctively the gap finding strategy tries to avoid the necessity of accurate distance measurements it was found that ultimately this strategy was still very much effected by them.
- *Difference between angle gap size and physical gap size*: This refers to the problem that a gap may be large enough for a robot to pass through but due to relative positioning may appear to have a very small angle gap, or vice versa.
- *Complex heuristics*: It was felt that the very complex heuristics used for gap selection would be hard to maintain, required a large learning curve, and that a small change or addition may break the entire heuristic base.

- *Limitations of local model:* Because this method does not use global path planning it comes with the inherent limitation of local obstacle avoidance models. A path to a destination may not be ultimately passable, i.e. a wall beyond locally known obstacles may be blocking the chosen path, or a chosen path may not be the most efficient when compared to considering all possible paths on a global scale.

The limitations with the Gap Finding model will be re-examined later in terms of how valid they still are in the new Best Gap system.

3.2.3 Stealth Dog

Stealth Dog was the name given to an obstacle avoidance skill used in 2003 and 2004 rUNSWift code (ref. [10]). The concept behind stealth dog in both years was relatively simple, though the implementation itself was somewhat more complex. The basic premise involves taking a curving path towards the ball instead of a straight path, with the curve being plotted so that it avoids nearby opponents.

Stealth Dog maintains a trajectory forty-five degrees away from the average of the headings of the two closest opponent dogs while heading to the ball. As the dog gets closer to the ball the heading to the opponent dogs will change in such a way that the dog traces a curve around the opponents to the ball. The stealth dog actions in both 2003 and 2004 were identical, the only thing that was changed was the trigger conditions. In both years conditions for executing Stealth Dog were very restrictive, the idea being that if the dog was uncertain it was better not to fire because stealthing increased the time taken to the ball.

Although the stealth dog skill has proved itself to be very effective in the past it is not feasible for the 2005 team because of the rewriting of the vision system. The GPS module no longer keeps track of the positions or headings of opponent dogs and instead only clouds of obstacle points are considered as will be explored in further detail in Chapter 4. Should opponent tracking be reinstated in the future, as no doubt it eventually would have been this year given more time, then stealth dog is a strategy a future rUNSWift team would be well advised to dust off and experiment with.

Chapter 4:

Obstacle Mapping

4.1 Obtaining Obstacle Points from Vision

Work on the rUNSWift vision module in 2005 was spearheaded by team member Alex North, readers are referred to his thesis report for a more thorough examination of the intricacies of the vision system (ref. [6]).

Every frame the vision module scans the current camera image for obstacles, which are detected on the basis of their shadows. All obstacles on the field cast a shadow and it was found to be more accurate to detect obstacle points based on these rather than identifying environment specific elements such as dogs, ball, beacons and so on.

Pixels are determined to be a shadow based on their Y value, if the Y value is below a certain threshold then the pixel is labeled as an obstacle. Because checking every pixel every frame would take more time than is available the vision system only checks pixels along vertical and horizontal scan lines that it traces across each image.

Obstacle points from scan lines are filtered to counteract excessive and spurious obstacles. For instance obstacle points directly below a large amount of green/field pixels are discarded as they are most likely spurious, this is because obstacle points are essentially shadows and if the shadow is below a clump of field that would seem to indicate that the field itself is casting a shadow.

As well, to limit excessive obstacle points only one point is taken from obstacle points found in clumps, and the number of points taken per scan line is limited to an upper threshold. As vertical scan lines are traced upwards from the bottom of the image the obstacle point threshold might seem to suggest that we would rarely see points that were further away as closer obstacle points would take up the quota for that scan line. This problem is overcome by having various length scan lines starting at differing positions in the image, as is evident in Image 1. Finally obstacle points that seem to be caused by the ball are discarded as we do not wish to consider the ball an obstacle or the dog might end up avoiding it.

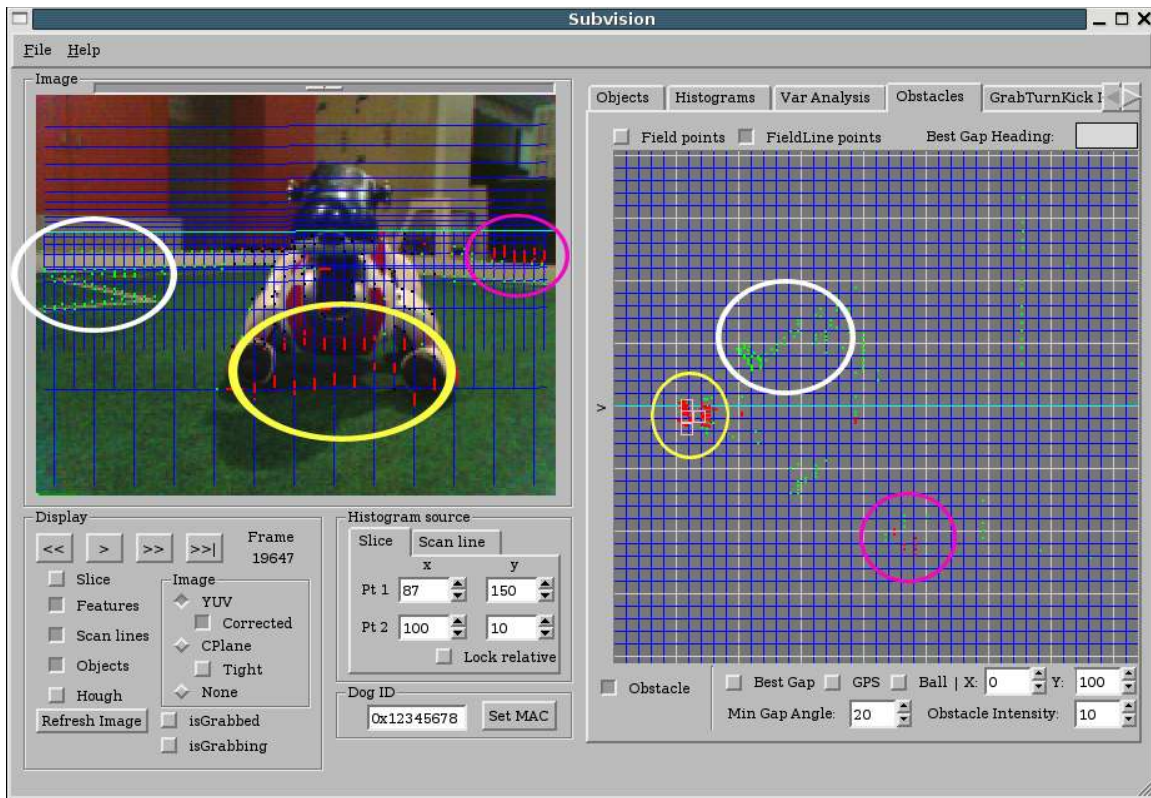


Image 1: LHS – Note the red obstacle points and green field line points picked up on the blue scan lines, note also the positioning and length of the scan lines on the image. RHS – Shows the mapping of the red obstacle points and green field line points into local coordinates where the dogs point of view is from the left side of the grid facing towards the grid.

The obstacle points are translated by the vision system into a local x,y,z coordinate and the points in this form are passed to the obstacle module.

4.2 Obstacle Probability Grid Mapping

Because of the prohibitive processing costs involved in working with obstacle points in their raw form the obstacle module stores obstacle points in an obstacle probability grid map. This is a grid covering, in potential, the entire field. Each grid square has an obstacle count associated with it, initially at zero. When an obstacle point is sighted in a given grid square the obstacle count for that square is incremented by one. A grid square can be decayed by multiplying it with a decay constant, D , which is a number between zero and one, how it is calculated shall be discussed further on. Due to the dynamic nature of obstacles on the field grid squares are decayed every frame, meaning that if an obstacle point is continually not sighted within a square then that square's obstacle count will decay to zero. Through the system of incrementation and decay the obstacle count of a grid square can be used as a measure of the probability that there is actually an obstacle at that position on the field.

In 2005 the inner playing field has dimensions of 360cm width and 540cm length, and we also want to consider obstacles 20cm outside the inner field since obstacles valid to play, such as beacons and goal walls, exist there. Using grid squares of 10cm^2 , the entire obstacle grid consists of 2128 squares as shown in Figure 9. Like working with raw obstacle points, working with such a large array also proves to be too intensive on the limited computation time and memory that is afforded every frame. However, due to obstacle probability decay and the limited range of robot sensors most squares in the grid are entirely empty for any given frame. With this in mind it is determined that only a percentage of the squares in the grid need actually be stored.

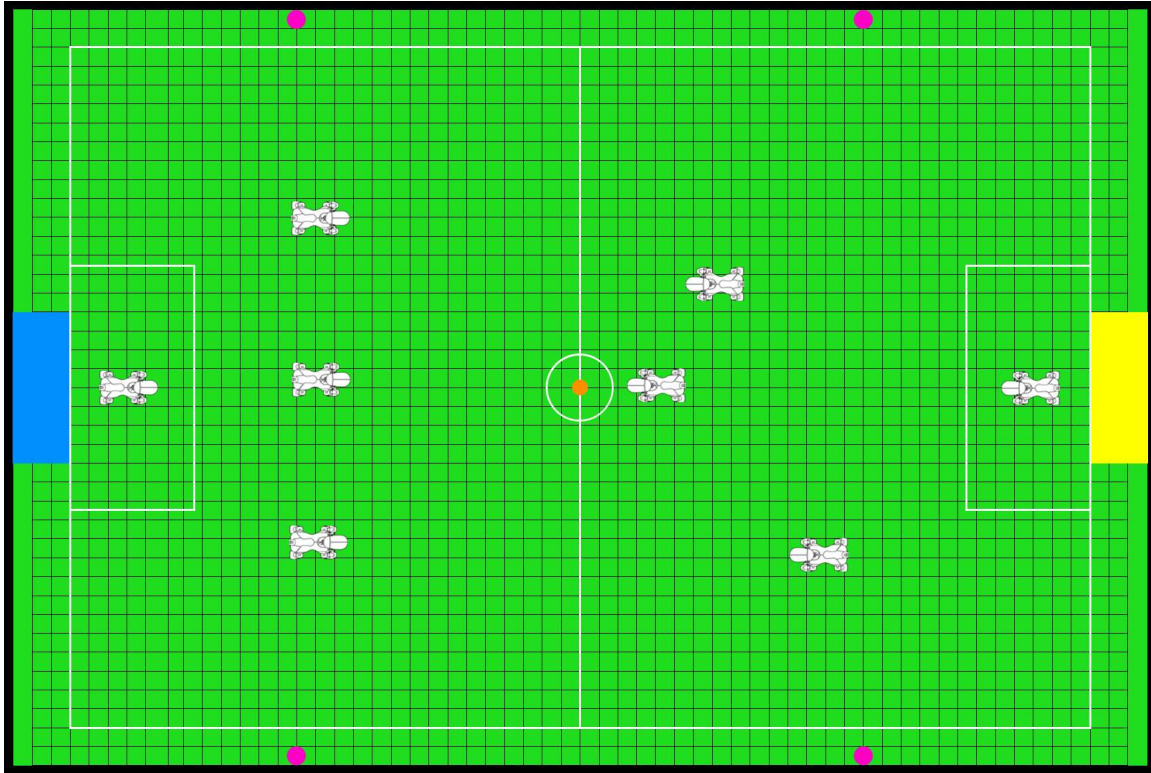


Figure 9: Obstacle Probability Grid overlaid on field

Due to obstacle decay keeping track of N grid squares is a relatively simple process. If a square decays to zero obstacle points it is discarded and if a point is found in a square that does not currently exist, and less than N squares are currently being tracked, then that square is added to those already being tracked. Obstacles in the grid tend to appear as clouds, with a small core of high probability grid squares surrounded by grid squares of lowering certainty.

In a perfect world an AIBO has a footprint in the obstacle grid of 6 squares, in reality since a dog is not looking down at the field from above it would generally take up half of that, around 6 to 10 boxes is still a good rough measure though because the part of the dog that is seen will be flanked by lower probability squares as fitting the obstacle cloud concept. Each dog can only see a maximum of 7 other dogs meaning that squares taken up by dogs should have a rough maximum of 70 squares. This is a generous estimation given the unlikeliness of having all other dogs in a single robots obstacle grid at one time.

Though dogs are the main obstacles we want to track, other objects are often relevant as well such as beacons, goal walls and the legs of referees walking across the field. Also, there is some amount of noise that comes in through the vision system if it mistakenly identifies obstacles where there are none. The amount of noise is generally dependent on how well the colour calibration has been done for the field and how different the current lighting conditions are to the ones when the colour calibration was done.

It has been observed that all the non-dog obstacles combined at any given time rarely account for as many boxes as are representing dogs at the same instant however in our calculations we assigned them an equivalent number of boxes to be safe. For 2005 rUNSWift assigned the value 200 as the maximum number of obstacle grid squares, roughly 10% of the possible maximum. This includes 70 grid squares for dog obstacles, 70 grid squares for non-dog obstacles and we round up to the nearest hundred to err on the side of caution. The extra grid squares also allow us to better account for a noisy field without losing too much accuracy.

Because a robot does not always know its location with a high degree of certainty it was found necessary to keep two separate obstacle probability grids in existence. One grid for local obstacles and one grid for global obstacles. When obstacle points are passed from the vision module to the obstacle module they are aligned into the local obstacle probability grid. At the beginning of every frame the local grid is cleared of any previous points so the local grid will always consist entirely of obstacle points seen that frame.

For each frame the obstacle module checks the GPS module to determine the certainty level regarding the robots current location on the field. If the certainty is above a predetermined level then the local obstacle points are converted to global coordinates and overlaid on the robots global obstacle probability grid, which is persistent between frames. Functions that perform operations over the obstacle probability grids may choose whether to use the local or global grid depending on location certainty, or on some separate criteria determined by the function.

The global obstacle grid consists of two other components apart from accumulated obstacle points passed from the vision module, namely constant obstacles and shared obstacles. Constant obstacles refer to known obstacles on the field, which in 2005 consists of four beacons and two goals (or four goal walls) as shown in Figure 10.

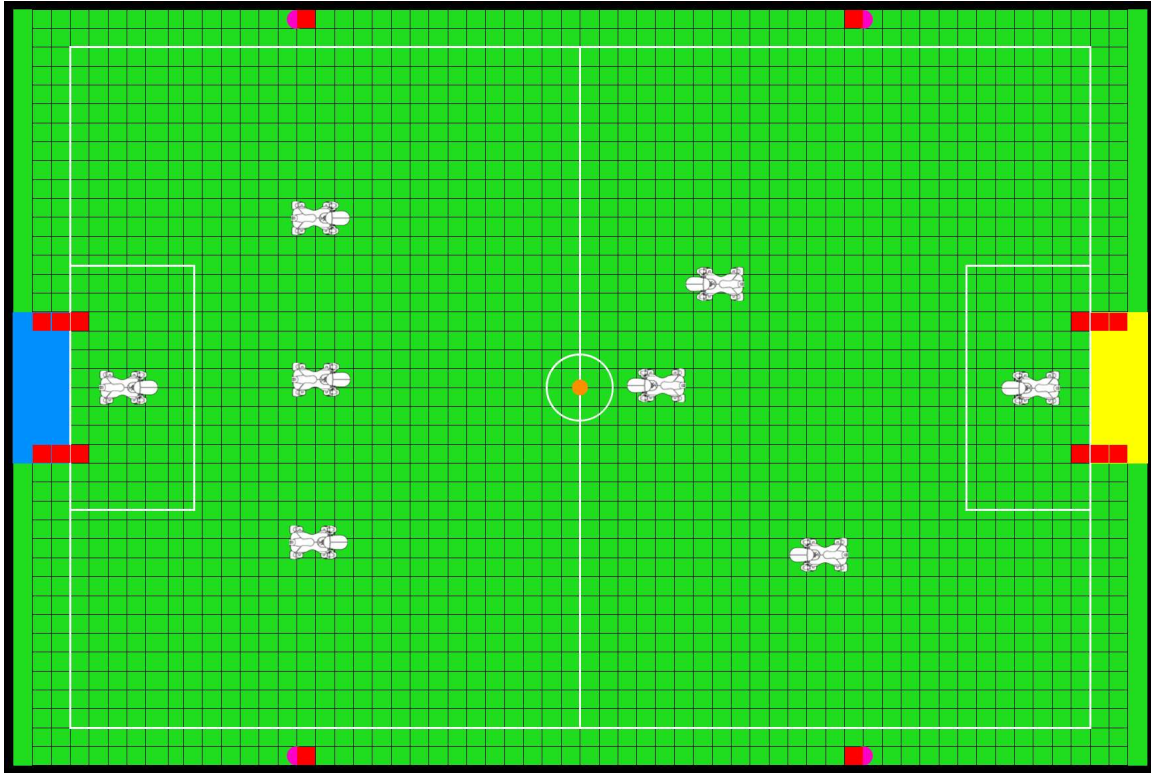


Figure 10: Obstacle Probability Grid with constant obstacles

Because obstacle points as passed from the vision system are not entirely reliable a probability grid mapping helps accounts for erroneous points. Probabilities decay every frame with a constant multiplier, derived mathematically (see Figure 11) to allow obstacles to remain in memory for a given head sweep of the environment. That is, if the robot does a continual sweep of the environment in front of it, obstacles should stay in the probability grid long enough for the robot to know the position of any obstacle within the scanning arc for the entirety of the time it takes to scan away and back again.

The obstacle decay constant is calculated using the formula shown below, which essentially calculates the decay constant necessary for an obstacle box to go from a high probability to zero in the time it takes for the dog to scan away and then scan back.

$$D^{2TF} \cdot X = Z$$

$$D^{2TF} = Z/X$$

$$D = (Z/X)^{\frac{1}{(2TF)}}$$

<i>Variable</i>	<i>Meaning</i>
X	minimum number of obstacle points in a grid square which is definitely an obstacle
T	time taken to scan the head from one side to the other e.g. from extreme left to extreme right
F	frames per second
Z	very small number
D	decay constant

Figure 11: mathematical derivation of the decay constant

4.3 Obstacle Box Sharing

Shared obstacles refer to obstacles boxes which are obtained from other robots in the team. Every frame each robot broadcasts its 24 most probable obstacle boxes and the other robots add these to their global obstacle probability grid. The limited number of boxes shared, less than 15% of possible maximum, helps counteract the inherent instability of passing global obstacles between robots with varying levels of certainty regarding their own position in the environment.

The shared obstacle count of each grid box is kept separate from the obstacle count as taken from the vision module. This allows functions that tap into the obstacle grids to specify whether or not to include shared obstacles. An example of this may be a function that requires a high amount of precision and cannot afford to rely on other robots observations, or a function which only cares about obstacles currently in direct line of sight and within sensor range.

Shared obstacles have their own decay constant which is set to a lower number than the normal obstacle decay meaning the shared obstacles will decay faster. This is to reflect how shared obstacles are inherently more unreliable since they were not seen by the current robot.

4.4 Problems and Limitations

Some issues that occur with this years obstacle mapping system are;

Array Traversal Runtime: Keeping the obstacle boxes in an array means that functions using the obstacle probability grids often must traverse the entire array to be able to calculate valid results. Sorting the array prior to traversal can be beneficial however this too requires extra runtime.

Grid Noise: The obstacle grid can be somewhat sensitive to noise, the sensitivity to noise will often depend on the colour calibration. For instance during the final matches in Osaka this year large crowds surrounded the field producing constant subtle varying of the lighting conditions on the field, as the vision module relies on shadows to detect obstacle points this was not ideal conditions for the obstacle module and was extremely difficult to calibrate effectively for.

Shared Boxes Runtime: Processing shared obstacle boxes takes up a lot of running time, often causing or related to frame dropping problems during the game.

Own Shadow: The greatest limitation of the obstacle mapping system this year is its inability to find obstacles closer than 50cm. This is an unfortunate side effect of using shadows for obstacle points in that up to 50cm we cannot determine if the obstacle point we are seeing is being caused by the dogs own shadow or by the shadow of an obstacle. Note that this doesn't mean there can't be any obstacles closer than 50cm away from the dog in the obstacle probability grid, it just means they either have to be a constant obstacle or a shared obstacle or they have to be seen by the dog further away and have the dog move closer to them before they decay.

4.5 Possible Solutions

To combat *array traversal runtime* obstacle grid squares could be stored in a hash table or some form of tree, it would most likely not be effective in all situations as a single key type would have difficulty accounting for the differing needs of all the functions accessing the obstacle grid. However it could attempt to be highly effective for the methods that access the grids most often. The downside to this solution is that it is a lot harder to implement and maintain.

Grid noise is a common phenomenon in any mapping system and generally the best that can be hoped for is to minimise the amount of noise. Inherently certainty grid based mapping systems already do a lot to filter out noise, an obstacle square that has a high obstacle count through spurious obstacle points will simply decay away to nothing when the dog repeatedly sees no more obstacle points in the box. As the obstacle points are based on shadows making sure the dogs are properly calibrated for the field and making sure they have good linear shifts is the another excellent way to combat this problem. For situations where it is known noise is a problem it is also advisable to adjust obstacle avoidance algorithm thresholds at which a grid squares' obstacle count indicates high probability.

Shared obstacle runtime is a somewhat tricky problem. The data is already being sent in an extremely compressed form and only a small amount of boxes are being traded. However 24 boxes per dog from 3 dogs adds up and they are each processed individually meaning 3 sets of overhead times. A possibility might be to, instead of trading the top 24 obstacle boxes, send obstacle boxes above a given probability threshold up to a maximum of 24. A faster wireless network may also help to alleviate this problem.

Own shadow is a very important problem to solve but also one of the most difficult. Attempts made this year to tighten criteria for obstacle points within possible own shadow met with limited success. One option may be to identify obstacles based on robot recognition rather than shadow recognition for own shadow regions.

Chapter 5:

Obstacle Avoidance

5.1 Path Planning

Initial attempts at obstacle avoidance this year looked into path planning by use of Rapidly-Exploring Random Trees commonly referred to as RRTs (ref. [13,14]). In this approach a grid is laid across the environment with obstacle grid squares designated as being non-passable. The current location and desired destination grid squares are marked and a path is generated between the two. The path is generated by sequential attempts to advance a potential path in either the direction to the destination or a random direction. For each path advancement attempt there is a probability R that the algorithm will attempt the random direction and the probability $1 - R$ that it will attempt the direction to the destination. The optimisation of the algorithm relies in no small part on determining an appropriate R value for the specific environment.

RRTs provide an alternative to path finding algorithms that sequentially iterate over possible options until they reach a viable path. Even algorithms that perform such sequential iterations intelligently still tend to balloon up exponentially in runtime. An RRT will often find a solution in an efficient manner however there is the trade off that you cannot guarantee a solution will be found in a finite amount of time.

Figures 12 and 13 highlight the random nature of the RRT algorithm, both are examples of paths planned with the same starting and ending points and the same set of obstacles. The resulting paths take much the same route, but the tree built during the search in example two is roughly double the size of the tree in example one.

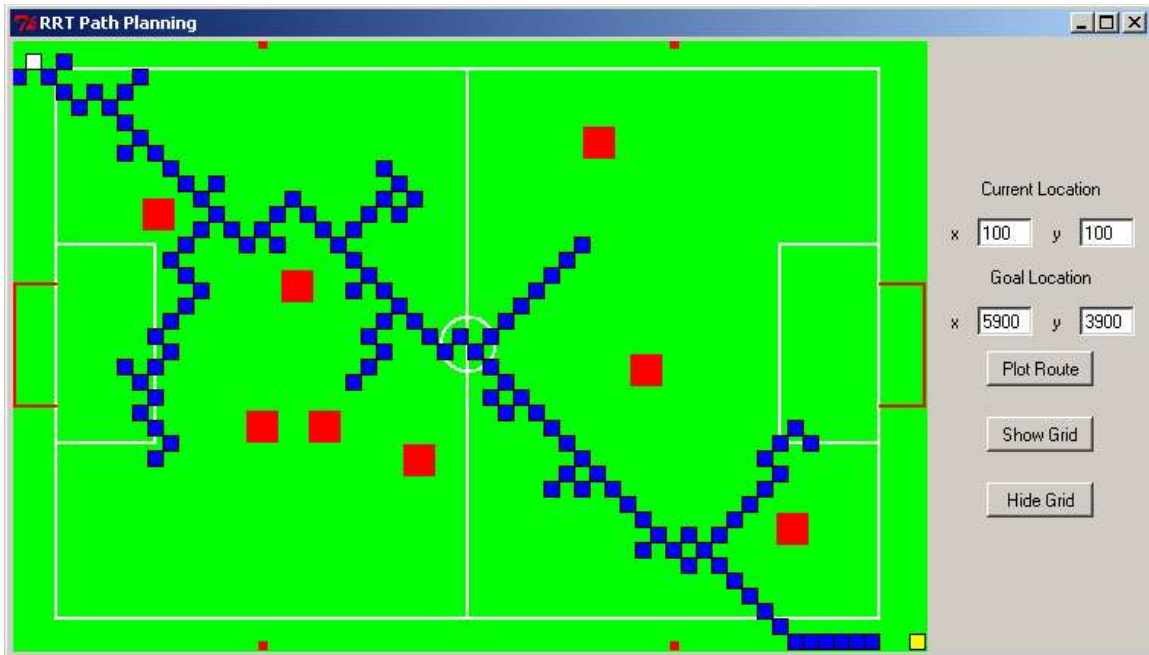


Figure 12: Example 1 of an RRT planned path, the explored tree has been left in to show methodology

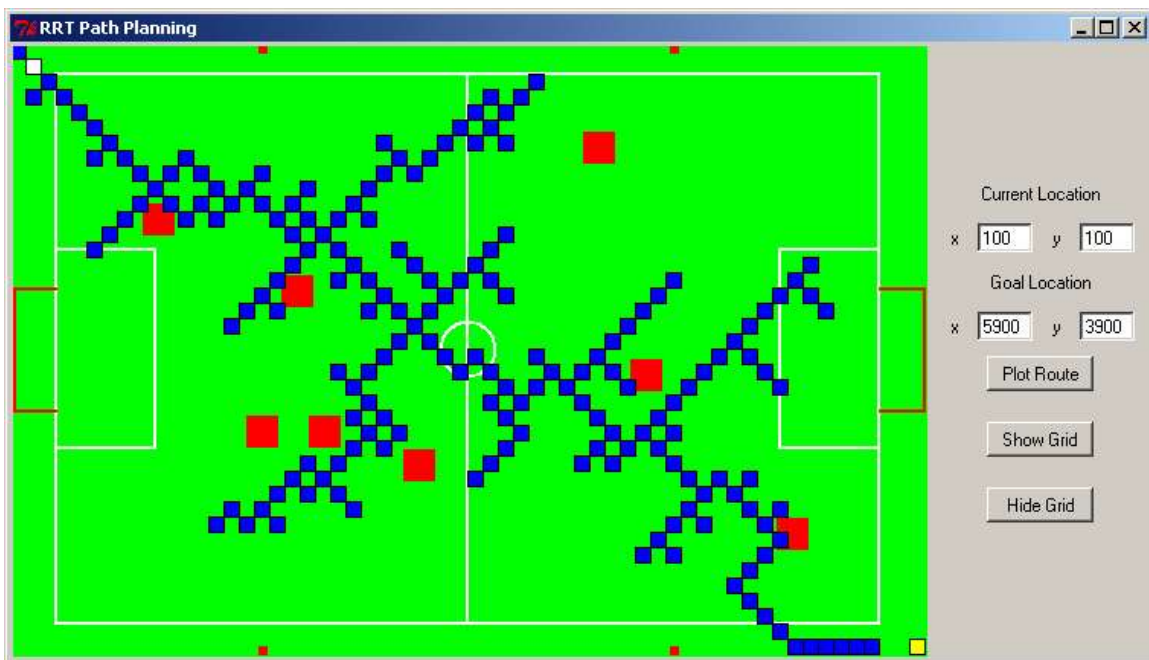


Figure 13: Example 2 of an RRT planned path, the explored tree has been left in to show methodology

A large series of sample paths through a simulated RoboCup field environment were generated with RRT using random obstacles and random start and end points, it was found that the average time to generate a path offline was 928ms, far above the frame time limit of 30ms, and on a machine with a lot more power than the AIBOs. If the RRT algorithm had been added to the code base the robot would have been constantly halting while trying to calculate new paths, not a good idea in a game of soccer.

Implementation of this algorithm up until this point had already taken a relatively long amount of time and though optimisation was considered it was decided the effort necessary to shave the algorithm down to a couple of milliseconds was too large, and it was uncertain if it was even possible, so the RRT method was abandoned. It is possible that there were programmer errors in the implementation or it could just be that the algorithm itself is not efficient enough to be a candidate for the high-speed requirements of the rUNSWift AIBO programming architecture.

5.2 Best Gap

Best Gap is a key algorithm in this years obstacle avoidance strategy. The idea behind Best Gap is that given the current location, heading and destination the algorithm will consult the obstacle grid and obtain the best gap for the robot to head to.

To calculate Best Gap the array of obstacle boxes is first sorted by their heading with regards to the robot. The algorithm then sequentially traverses the array and tests the gap between each obstacle box, when the last box is reached the gap between the last box and the first box is tested. A valid gap must be greater than a given minimum gap size in degrees and the best gap is the valid gap containing the heading closest to the heading to the destination. As the array is sorted by heading, traversing the array is equivalent to scanning a 360 degree circle around the dog.

The Best Gap Algorithm runs in the following manner (see Figure 14 for a pseudocode interpretation of the algorithm);

Step 1) Filter out obstacle boxes that do not meet criteria.

These include boxes below a specified obstacle point threshold, boxes closer than a specified distance or boxes further away than a specified distance.

As the algorithm only considers the heading of an obstacle box beyond this point it is also useful to filter out obstacle boxes with the same heading as an already existing box. For greater clarity Figure 15 gives a visual representation of the filtering criteria.

Step 2) If there are no valid obstacle boxes left then return the direct heading to the destination.

Step 3) If there is only one valid obstacle box then return a heading as direct as possible to the destination, while making sure the heading is in a gap of at least the specified minimum gap size.

Step 4) Sort the obstacle box array by heading.

Step 5) Traverse the array, test the gap between each subsequent obstacle to see if it is greater than or equal to the specified minimum gap size.

If the gap is of sufficient size find the closest heading to the destination in the gap, if the heading is closer than any previously found heading mark the gap as the current favourite.

Step 6) Test the gap between the last obstacle box and the first obstacle box as in Step 5.

PSEUDOCODE FOR BEST GAP CALCULATION

```
FOR i = 0 TO numBoxes - 1
    IF DoesNotMeetCriteria(boxes[i]) THEN
        DeleteFromArray(boxes, i);

##Special Case: no valid obstacle boxes
IF numBoxes = 0 THEN
    RETURN HeadingToDest;

##Special Case: there is only one valid obstacle box
IF numBoxes = 1 THEN
    RETURN BestHeadingToDestAvoidingObstacle(boxes[0]);

boxes = QuicksortOnHeading(ObstacleBoxes);

##Find the best gap by checking the gap between each obstacle box
bestGapHeading = LARGE_NUM;
FOR i = 1 TO numBoxes - 1
    IF GapBetween(boxes[i], boxes[i-1]) >= MinimumGap THEN
        heading = HeadingToDestInGap(boxes[i], boxes[i-1]);
    IF ABS(heading - HeadingToDest) <
        ABS(bestGapHeading - HeadingToDest) THEN
        bestGapHeading = heading;

##Test the gap between the last box and the first box
IF GapBetween(boxes[0], boxes[numBoxes - 1]) >= MinimumGap THEN
    heading = HeadingToDestInGap(boxes[0], boxes[numBoxes - 1]);
    IF ABS(heading - HeadingToDest) <
        ABS(bestGapHeading - HeadingToDest) THEN
        bestGapHeading = heading;

RETURN bestGapHeading;
```

Figure 14: Pseudocode for Best Gap Algorithm

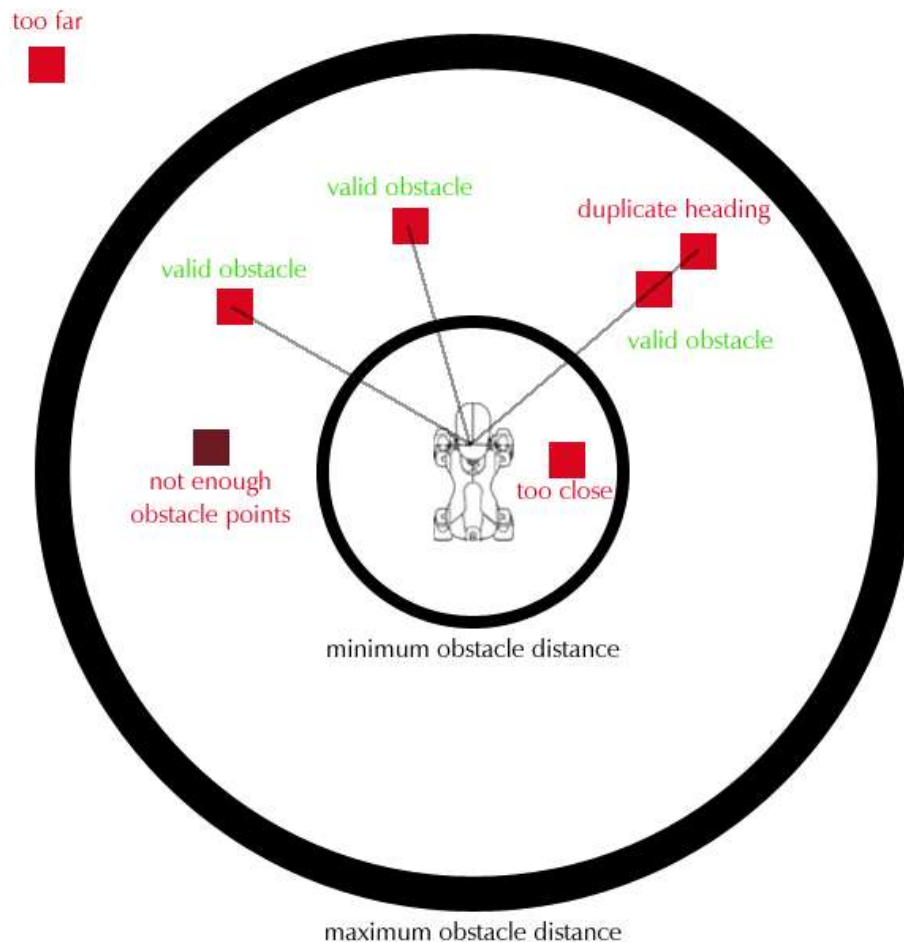


Figure 15: Visual representation of Best Gap filtering criteria

Unfortunately it is not uncommon for Best Gap to suffer from extreme fluctuations between frames. The algorithm itself in a single instant is very stable, it will look at the available data, ask you where you want to go and tell you exactly which way you should head to get there. However as illustrated in Figure 16 at one instant you may wish to go left around an obstacle and then the next instant you may wish to go right to avoid it, a robot solely using Best Gap can often end up 'fish tailing' between left and right. This can be caused by several things including noise in the obstacle grid or variance in the localisation module, that is your idea of your position may be different to your actual position. Obviously this is extremely undesirable as the greater a heading is switched between frames the more momentum is lost and the longer it will take the dog to get to its destination.

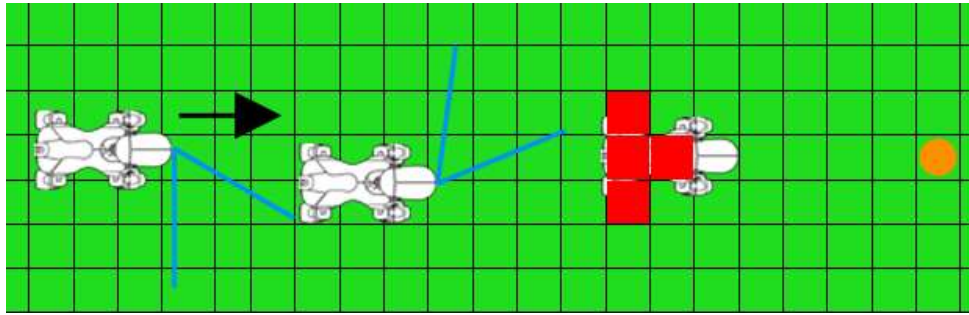


Figure 16: illustration of 'fish tailing'

In order to get around this problem a higher level strategy needed to be built on top of the Best Gap algorithm, something that could add hysteresis and in general be aware of game play issues other than the need to avoid the nearest obstacles.

5.3 High-Level Strategy – Dodgy Dog

Every frame the system will use its current position and current role to decide on a desired destination. Chosen destinations can be varied, a defensive position, the current known location of the ball, a search pattern if the ball has been lost, or one of several other considerations. Whatever the destination the obstacle avoidance module is engaged to give a heading and speed along which to continue from the current position to get there, taking into account the known locations of obstacles.

In the top level behaviour a new skill was added this year to perform obstacle avoidance. This new skill is named Dodgy Dog, and at its core is the Best Gap algorithm discussed in the previous section. As discussed, a major problem of using Best Gap in its raw state is the fluctuating path that can be produced as a result. Dodgy Dog employs several strategies in an attempt to smooth out a path created by a sequence of instance best gaps.

The first feature of Dodgy Dog is being able to tell when it should or should not be deployed. Before using Dodgy Dog another behaviour level system may first query it, telling it where it wants to go and asking if it is appropriate to use Dodgy Dog to get there. To evaluate the answer Dodgy Dog looks at the distance to the destination and how many obstacles are in the corridor from the dog to the destination. If the destination is closer than a predefined minimum dodging distance Dodgy Dog will recommend not dodging. If the destination is far enough away then Dodgy Dog will still only recommend deployment if there is a minimum threshold of obstacles in the corridor to the destination, and it will only consider obstacles up to a maximum distance away.

Hysteresis is perhaps the most important feature Dodgy Dog adds to Best Gap. The best heading found this frame is combined with hysteresis information on the best heading used last frame to determine movement. The hysteresis formula used is

$$\frac{(N * \text{Best Heading}) + (P * \text{Best Heading Last Frame})}{N + P}$$

$$N + P$$

Where N and P were determined experimentally by testing various value pairs and seeing how well the dog could dodge around an obstacle on a straight path in controlled conditions using the values. The graph in Figure 17 shows the results from experimentation with N and P becoming 3 and 1 respectively.

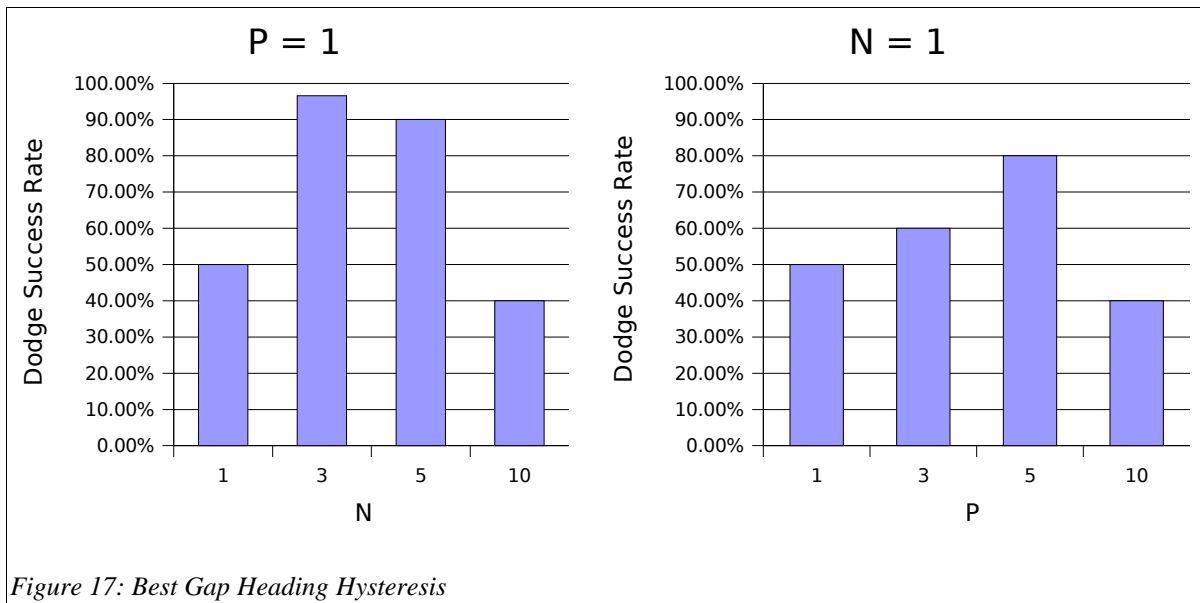


Figure 17: Best Gap Heading Hysteresis

Note that the 'best heading last frame' value of the hysteresis formula includes previous hysteresis information and is not just the raw value obtained from Best Gap last frame.

Dodgy Dog also limits how far the dog may attempt to adjust its heading per frame, this further helps to smooth out jerkiness in the path. This will not effect a necessary large heading adjustment though, it just means the heading adjustment will be broken up into multiple smaller adjustments over a period of time.

Finally, though Best Gap will only return a heading to avoid obstacles, if Dodgy Dog detects that obstacles are very close it will add a side step to the robots walk in the direction of the robots heading relative to the destination heading as illustrated in Figure 18. Note that in the diagram the robot will sidestep left even though its heading is going to the right. This is because its heading is to the left of the destination heading, if the dog were to sidestep right in this situation there is a good chance it would sidestep straight into the path of an obstacle. Avoiding an obstacle by sidestepping is better for close obstacles as the robot may not be able to turn fast enough to follow the best gap heading that avoids it.

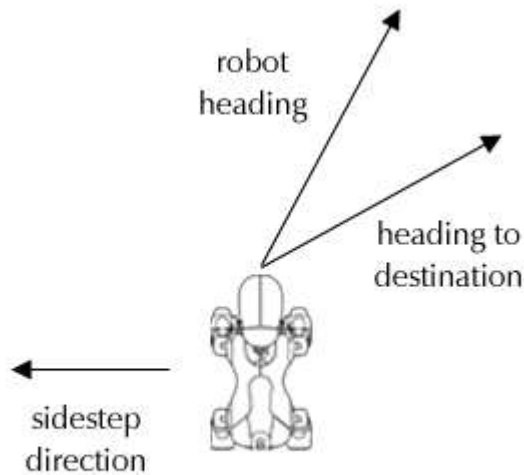


Figure 18: Dodgy Dog sidestep direction

Dodgy Dog, using Best Gap, performed admirably in competition contributing in no small part to rUNSWifts placing of third in the finals.

5.4 Problems and Limitations

The RRT path planning examined in the first section of this chapter turned out to be a failure due to expensive runtime. Time limitations for programming an obstacle avoidance system meant this issue was not fully explored however we can speculate the large runtime may have been due to some inefficient coding, errors in the way the algorithm was implemented or merely a resistance in the algorithm to scale down to such a small processing footprint.

There is some inherent error in the Best Gap algorithm in that it does not consider the distance to or between obstacles beyond filtering thresholds and just relies on their heading from the dog. This is a problem because while angles are good in theory as a measure of a gap, they can be deceiving when related to distance as illustrated in Figure 19. As discussed earlier this was an issue with the 2003 Gap Finding strategy as well and was one of the reasons, albeit a minor one, that they did not use this strategy. This problem is less of an issue this year however because, like many of the issues in Gap Finding, the negative effects are limited by the dynamic obstacle environment. In the 2003 challenge if they did not detect a path between obstacles because of a angle/distance relation then that was a viable path that was being rejected, possibly even the best path, in real game play however paths are changing all the time and plotting a route between two moving dogs is in most cases not the best of ideas anyway. The filtering distance thresholds help to relieve the worst effects of this problem but this is still an area where improvements could be applied.

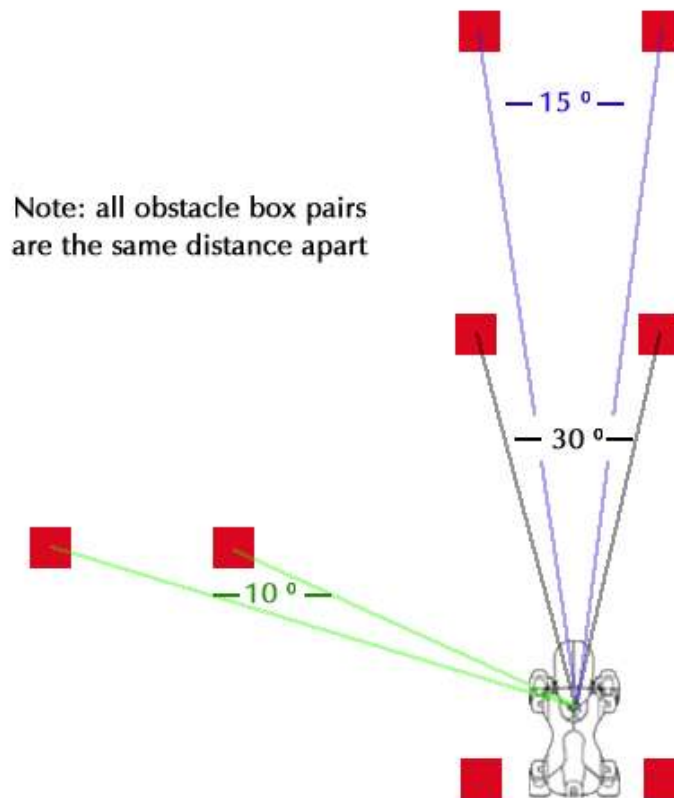


Figure 19: difference between distances and angles

Gap Finding had three other limitations that caused it to be dismissed, namely reliance on obstacle distances, complex heuristics and limitations of the local model. Reliance on obstacle distances was an issue in Gap Finding because they could not get reliable measurements to dogs from their vision system however with the new vision system this year distances to obstacles are more reliable, furthermore distances needed to be very exact in the challenge because even the smallest touch with another robot carried a heavy penalty whereas in the real game it doesn't matter as much. Complex heuristics have not been a problem in Best Gap or Dodgy Dog, and the heuristics for each are not as interconnected as they were in Gap Finding, meaning that changing a single heuristic will not break all the others.

The last of the Gap Finding problems was local model limitations, or the inherent limitations of a local versus a global obstacle avoidance model. This is once again more relevant to the challenge than the real game as the challenge had a static environment and the robot had more time to evaluate where it was and where it should go. The global obstacle path planning model lends itself more to this type of setup because you can get an advantage taking the time to plot a route through the static maze. These advantages do not carry over well to the real game environment however because of its dynamic nature, an algorithm that is faster, even at the cost of accuracy, can still do much better. In fact though it may seem counter intuitive in some situations the cost of accuracy is actually a bonus. This is because the obstacles themselves are mobile and sometimes just heading straight and pushing a way through to the ball is the best strategy to follow.

The new vision system this year is a remarkable piece of work however there was not time before the tournament to add in the ability to distinguish robots of different teams. Because of this, unlike previous years, the GPS module did not contain information on current position and headings of opponent dogs. This can act as a limitation to the obstacle module and hinder the development of more advanced obstacle avoidance strategies.

5.5 Possible Solutions

The problem of gap angle versus gap size could be addressed with formulas to take into account actual gap size as well as the gap angle, the maths to calculate this is not too complex however the heuristics of working with the gap sizes would be tricky to add to the existing system. In general this is not a very large problem and the time needed to address it could be spent far better elsewhere, though it may be worth looking at should the field size become larger at some point in the future which would make the angle versus distance more pronounced.

As mentioned local model limitations as opposed to global path planning do not seem to be such a big issue, however if future team members are interested the floor is wide open for a fresh attempt at a path planning algorithm.

No doubt next year one of the first things the new team next year will wish to do is re-implement the ability of the vision module to distinguish dogs and thus the ability of the GPS module to keep track of them. If this is successfully implemented it opens up several possibilities. The Best Gap algorithm currently treats both friend and opponent as the same type of obstacles out of necessity but if it could tell the difference then in situations where it was avoiding friendly dogs it could decrease its avoidance heading if the friendly dog agreed to sidestep out of its path. For example, instead of the avoiding dog going 30 degrees to the left, it could go 15 degrees to the left if the friendly dog being avoided agreed to sidestep 20cm out of the path of the avoiding dog, keeping them both closer to the target.

It may seem like this is possible now since the GPS module keeps track of team member locations, however it would be extremely easy to misclassify an obstacle cloud as a single dog when it is actually a team member and an opponent clumped together, which would throw off the strategy. This is just one of many possible teamwork strategies that could be implemented knowing which obstacle clouds were enemies.

Chapter 6:

Conclusion

This thesis report went into details over the advancements made by the rUNSWift team in 2005 in the fields of Linear Shifting and Obstacle Mapping and Avoidance while coding to participate in the world finals in Osaka, Japan.

The rUNSWift vision module relies on exact YUV measurements to classify the colours of objects on the field, the ranges of YUV values for given object colours are so limited that the minute differences between the cameras on separate dogs can effect object classification. Since 2004 rUNSWift have used a linear shift table to better apply a colour calibration across the varying dogs, this year a program was created that automates a lot of the labour necessary to calculate the table. Furthermore a program was developed to painlessly reconfigure an existing shift table for a new base dog with relatively little error, though it is recommended to calculate the table from scratch where possible. Both these systems advance the ability of rUNSWift team members to control in great detail the overall vision calibration of the AIBO dogs and give them an edge in game play.

A sophisticated Obstacle Probability Grid has been built that combines known information about the environment (constant obstacles), shared information about the environment (shared obstacles) and observed information about the environment (vision obstacles) to create an internal world model that is as accurate as possible.

An attempt was made at a global path planning strategy in the form of Rapidly-Exploring Random Trees however the attempt was abandoned due to large runtime requirements. A local obstacle avoidance strategy, Best Gap, was developed and worked admirably when combined with its higher level counterpart of Dodgy Dog, the new behavioural skill that has been added to the rUNSWift dogs' repertoire. Both components of the developed obstacle avoidance strategy have relatively simple concepts at their core. Best Gap scans a 360 degree arc around the dog and determines the best gap to head into to get to the destination. Dodgy Dog adds intelligence to Best Gap, using heuristics and hysteresis to smooth out the obstacle avoidance path resulting from a series of Best Gap calls over time.

Global path planning versus local obstacle avoidance is an ongoing issue in the RoboCup world, on one hand Global Path planning should allow you to get an optimum path to your destination without getting stuck in local dead ends. On the other hand the incredibly dynamic nature of obstacles on the field (getting faster every year) means that a best path will not remain the best path for long, and local dead ends likewise have a way of clearing up pretty quickly. The failure of this years attempt at path planning cannot be taken as a mark against global path planning however the success of this years implementation of local obstacle avoidance shows its power in such dynamic environments.

The main limitation of this years Obstacle Mapping and Avoidance system is its inability to deal with obstacle points closer than 50cm, for scrums and other situations where a robot may be stuck up against other dogs it is often necessary to wait for the other dogs to change positions before the robot can disengage itself. Although this is an undesirable flaw the obstacle module as a whole still performs very well in competition. However this is still the number one issue that must be dealt with in future uses of this strategy.

Bibliography

[1] *RoboCup Web Site 2005*

Site: <http://www.robocup.org>

[2] *RoboCupSoccer Four-Legged League Rules 2005*

From Site: <http://www.tzi.de/4legged/bin/view/Website/WebHome>

Direct Link: <http://www.tzi.de/4legged/pub/Website/Downloads/Rules2005.pdf>

[3] *RoboCupSoccer Four-Legged League Rules 2004*

From Site: <http://www.tzi.de/4legged/bin/view/Website/History>

Direct Link: <http://www.tzi.de/4legged/pub/Website/History/Rules2004.pdf>

[4] *AIBO SDE Web Site, OPEN-R*, <http://openr.aibo.com/>

[5] *Sony AIBO ERS-7 User Guide*, Sony

[6] North A., *Object Recognition from Sub-Sampled Image Processing*, The University of New South Wales, School of Computer Science & Engineering, September 2005

[7] Morioka N., *Road To RoboCup 2005: Behaviour Module Design and Implementation, System Integration*, The University of New South Wales, School of Computer Science & Engineering, September 2005

[8] Sianty A., *Honours Thesis*, The University of New South Wales, School of Computer Science & Engineering, September 2005

[9] Chen W.M., *Honours Thesis*, The University of New South Wales, School of Computer Science & Engineering, September 2005

[10] Chen J., Chung E., Edwards R., Wong N., Hengst B., Sammut C., and Uther W., *Rise of the AIBOs III – AIBO Revolutions*, The University of New South Wales, School of Computer Science & Engineering, November 4 2003

From Site: <http://www.cse.unsw.edu.au/~robocup/2005site/reports.phtml>

Direct Link: <http://www.cse.unsw.edu.au/~robocup/2003site/report2003.pdf>

[11] Mak E., *Real-Time Obstacle Avoidance and Path Planning for Mobile Robot with Limited Onboard Processing Power*, The University of New South Wales, School of Computer Science & Engineering, October 29 2003

- [12] Xu J., 2004 *rUNSWift Thesis – Low Level Vision*, The University of New South Wales, School of Computer Science & Engineering, September 2004
From Site: <http://www.cse.unsw.edu.au/~robocup/2005site/reports.phtml>
Direct Link: <http://www.cse.unsw.edu.au/~robocup/2004site/rUNSWift2004.tgz>
- [13] Bruce J., and Veloso M., *Real-Time Randomized Path Planning for Robot Navigation*, Proceedings of IROS-2002, Switzerland, October 2002
- [14] Branicky M.S., Curtiss M.M., Levine J.A., and Morgan S.B., *RRTs for Nonlinear, Discrete, and Hybrid Planning and Control*, Proceedings of the 42nd IEEE Conference on Decision and Control, Vol. 1, Maui, Hawaii USA, December 9-12 2003, pp. 657-663
- [15] Keith J., *Video Demystified: A Handbook for the Digital Engineer, Second Edition*, ISBN: 1-878707-23-X (paperbound), ISBN: 1-878707-36-I (casebound), Printed in the United States of America, Copyright 1996 by HighText Interactive, Inc., San Diego, CA 92121
- [16] Weisstein E.W., *Least Squares Fitting*, MathWorld – A Wolfram Web Resource
Site: <http://mathworld.wolfram.com/LeastSquaresFitting.html>
- [17] Elfes A., *A Sonar-Based Mapping and Navigation System*, 1986 IEEE International Conference on Robotics and Automation, Vol. 3, April 1986, pp. 1151-1156
- [18] Moravec H.P., and Elfes A., *High resolution maps from wide angle sonar*, IEEE International Conference on Robotics and Automation Proceedings, Vol. 2, March 1985, pp. 116-121
- [19] Borenstein J., and Koren Y., *Real-Time Obstacle Avoidance for Fast Mobile Robots*, IEEE Transactions on Systems, Man and Cybernetics, Vol. 19, No. 5, September/October 1989, pp.1179-1187
- [20] Khatib O., *Real-time obstacle avoidance for manipulators and mobile robots*, IEEE International Conference on Robotics and Automation Proceedings, Vol. 2, March 1985, pp. 500-505
- [21] Krogh B.H., *A generalized Potential Field Approach to Obstacle Avoidance Control*, International Robotics Research Conference, Bethlehem, Pennsylvania, August 1984.
- [22] Borenstein J., and Koren Y., *The Vector Field Histogram – Fast Obstacle Avoidance For Mobile Robots*, IEEE Journal of Robotics and Automation Vol. 7, No. 3, June 1991, pp. 278-288

[23] Borenstein J., and Ulrich I., *VFH+: reliable obstacle avoidance for fast mobile robots*, IEEE International Conference on Robotics and Automation Proceedings, Vol. 2, May 16-20 1998, pp. 1572-1577

[24] Press W.H., Teukolsky S.A., Vetterling W.T., Flannery B.P., *Numerical Recipes in C: the art of scientific computing, Second Edition*, ISBN: 0-521-43108-5, Cambridge University Printing Press, Printed in the United States of America, originally published 1992, pp. 656-699

Appendices

The rUNSWift Team

The University of New South Wales rUNSWift team is the only team in the Four-Legged League that replaces all its members every year. All members in the rUNSWift team are undergraduates doing RoboCup as a major project or thesis and in their fourth year. This years team consisted of Wei Ming Chen, Nobuyuki Morioka, Alex North, Joshua Shammay and Andrew Sianty. Two volunteer coders who were not on the team but whose help was greatly appreciated were Derek Leung and Phu Quang Victor Phung.

The high turnover rate of team members has the disadvantage of the team each year needing to relearn a lot of base material essential to working with the team code base. However this is greatly offset by the influx of new ideas and new ways of thinking that come with an entirely new generation of RoboCup participants. The rUNSWift team was established in 1999 and has been the most successful team in the league to date.

<i>Year</i>	<i>Placing</i>
1999	2 nd
2000	1 st
2001	1 st
2002	2 nd
2003	1 st
2004	¼ Finalists
2005	3 rd

*Table 3: Placings
achieved by rUNSWift*

Programming Code for Dodgy Dog

Written in Python

```
#
# Copyright 2005 The University of New South Wales (UNSW) and National
# ICT Australia (NICTA).
#
# This file is part of the 2005 team rUNSWift RoboCup entry. You may
# redistribute it and/or modify it under the terms of the GNU General
# Public License as published by the Free Software Foundation; either
# version 2 of the License, or (at your option) any later version as
# modified below. As the original licensors, we add the following
# conditions to that license:
#
# In paragraph 2.b), the phrase "distribute or publish" should be
# interpreted to include entry into a competition, and hence the source
# of any derived work entered into a competition must be made available
# to all parties involved in that competition under the terms of this
# license.
#
# In addition, if the authors of a derived work publish any conference
# proceedings, journal articles or other academic papers describing that
# derived work, then appropriate academic citations to the original work
# must be included in that publication.
#
# This rUNSWift source is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
# General Public License for more details.
#
# You should have received a copy of the GNU General Public License along
# with this source code; if not, write to the Free Software Foundation,
# Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

#
# Copyright (c) 2005 UNSW
# All Rights Reserved.
#

#=====
=====
# Python Behaviours : 2005 (c)
#
```

sDodgyDog - edited by Joshua Shamma.

#=====

import Global
import Constant
import VisionLink
import Debug
import Action
import Indicator

import hMath
import sFindBall

GAP_MAX_DIST = -1
BEST_GAP_DIST_LIMIT = 80

MIN_BOX_INTENSITY = 30
MIN_DODGY_TARGET_DIST = 25 # Don't dodge if closer to destination than this
CLOSE_DODGY_OBSTACLE_DIST = 50 # Add sidestep when this close to an obstacle
MAX_DODGY_OBSTACLE_DIST = 150 # Don't dodge any obstacle farther than this
MAX_DODGY_DODGE = 60 # Don't divert by more than this (degrees)
CORRIDOR_WIDTH = 40
CLOSE_CORRIDOR_WIDTH = 15
MIN_OBS_IN_CORRIDOR = MIN_BOX_INTENSITY * 2

FIXED_DEST_VAR = 20

SIDESTEP_FOR = 10 #number of frames to sidestep when close to an obstacle

gUseMinGap = 30 # Min gap size (degrees)
gUseMinIntensity = MIN_BOX_INTENSITY # Min obs intensity to be considered
gUseMinDist = 10 # Min obs distance to be considered

#hysteris
gNewHeading = True
gLastBestHeading = 0
gBestHeadingInfluence = 3.0
gLastBestHeadingInfluence = 1.0
gSidestepCounter = 0
gLeft = 0

def DecideNextAction():
 dodgyDogToBall()


```

# True if there are obstacles between me and ball that require dodging
def shouldIBeDodgyToBall():
    if Global.vBall.isVisible():
        ball = Global.vBall.getPos()
    else:
        ball = Global.gpsGlobalBall.getPos()

    if Global.lostBall > Constant.LOST_BALL_GPS:
        return False

    return shouldIBeDodgy(ball[0], ball[1])

# True if there are obstacles along heading (local, zero right) that require
# dodging
def shouldIBeDodgyAlongHeading(heading):
    # Pick a point out on the heading and dodgy to that point
    heading = hMath.local2GlobalHeading(Global.selfLoc, heading)
    pos = hMath.polarToCart(100, heading)
    #print "shouldIBeDodgyAlongHeading", heading, pos
    return shouldIBeDodgy(pos[0], pos[1], False)

# True if there are obstacles between me and destX/Y (global) that require
# dodging
def shouldIBeDodgy(destX, destY, fixedDest = False,\
    fixedDestVar = FIXED_DEST_VAR):
    global gUseMinDist, gUseMinIntensity

    selfX, selfY = Global.selfLoc.getPos()
    selfH = Global.selfLoc.getHeading()
    dist = hMath.getDistanceBetween(selfX, selfY, destX, destY)
    localDest = hMath.getLocalCoordinate(selfX, selfY, selfH, destX, destY)
    localHead = hMath.cartToPolar(*localDest) # zero right

    if dist <= MIN_DODGY_TARGET_DIST:
        return False

    if fixedDest and dist < fixedDestVar:
        return False

    # clip distance we count obstacles to MAX_DODGY_OBSTACLE_DIST or
    # (dist - MIN_DODGY_TARGET_DIST) so we don't dodge obstacles that are
    # miles away, and don't dodge obstacles that are right next to the
    # target.
    dist -= MIN_DODGY_TARGET_DIST
    if dist >= MAX_DODGY_OBSTACLE_DIST:

```

```

    dist = MAX_DODGY_OBSTACLE_DIST
    localDest = hMath.polarToCart(dist, localHead)

    tmp = VisionLink.getNoObstacleBetween(0, 0,
                                           int(localDest[0]), int(localDest[1]),
                                           CORRIDOR_WIDTH, gUseMinDist,
                                           gUseMinIntensity, Constant.OBS_USE_NONE)

    if Debug.dodgyDebug:
        print "shouldIBeDodgy: obs in corr lhead, dist (" , localHead, \
            " , dist, ") = " , tmp,

    if tmp > MIN_OBS_IN_CORRIDOR:
        if Debug.dodgyDebug:
            print "-> True"
        return True

    if Debug.dodgyDebug:
        print "-> False"
    return False

#go to ball, dodge obstacles on the way
def dodgyDogToBall():
    global gNewHeading

    if Global.lostBall > Constant.LOST_BALL_GPS:
        if Debug.dodgyDebug:
            print "dodgyDogToBall: cannot see ball, searching"
        gNewHeading = True
        return sFindBall.perform()

    if Global.vBall.isVisible():
        ball = Global.vBall.getPos()
        #headAction = hFWHead.mustSeeBall
    else:
        ball = Global.gpsGlobalBall.getPos()
        #headAction = hFWHead.mustSeeGpsBall

    rtn = dodgyDogTo(ball[0], ball[1])

    #look at ball
    #hFWHead.compulsoryAction = headAction
    #hFWHead.DecideNextAction()

    return rtn

```

```

# Go along specified heading (local, zero right), dodging obstacles on the way
def dodgyDogAlongHeading(heading):
    # Pick a point out on the heading and dodgy to that point
    heading = hMath.local2GlobalHeading(Global.selfLoc, heading)
    pos = hMath.polarToCart(100, heading)
    return dodgyDogTo(pos[0], pos[1], False)

#go to destination, dodge obstacles on the way
#destination is specified in global co-ordinates
#set fixedDest to true if you are moving towards
#a set point on the field (as opposed to a moving point/object)
def dodgyDogTo(destX, destY, fixedDest = False, fixedDestVar = FIXED_DEST_VAR):

    global gLastBestHeading, gNewHeading
    global gBestHeadingInfluence, gLastBestHeadingInfluence
    global gUseMinGap, gUseMinIntensity, gUseMinDist
    global gSidestepCounter, gLeft

    myPos = Global.selfLoc.getPos()
    myHeading = Global.selfLoc.getHeading()
    localDest = hMath.getLocalCoordinate(myPos[0], myPos[1], myHeading,
                                         destX, destY)
    localHead = hMath.cartToPolar(*localDest) - 90 # zero forward
    distToDest = hMath.getDistanceBetween(myPos[0],myPos[1],destX,destY)

    if fixedDest and distToDest < fixedDestVar:
        return Constant.STATE_SUCCESS

    # bestGap = (hLeft, hRight, hBest, gapseize) degrees
    bestGap = VisionLink.getBestGap(localDest[0], localDest[1],
                                    GAP_MAX_DIST, gUseMinDist, gUseMinGap,
                                    gUseMinIntensity, Constant.OBS_USE_NONE)

    if bestGap != None:
        if Debug.dodgyDebug:
            print "Dodgy: localxy_dest", localDest, "dist", distToDest
            print "bestgap (left, right, best, size) =", bestGap

        thisBestHeading = bestGap[2]

    # Don't try to head more than x away from direct heading
    if abs(localHead - thisBestHeading) > MAX_DODGY_DODGE:
        if thisBestHeading > localHead:
            thisBestHeading = localHead + MAX_DODGY_DODGE
        elif thisBestHeading < localHead:
            thisBestHeading = localHead - MAX_DODGY_DODGE

```

```

if gNewHeading:
    gLastBestHeading = thisBestHeading
    gNewHeading = False

bestHeading = thisBestHeading * gBestHeadingInfluence
bestHeading += gLastBestHeading * gLastBestHeadingInfluence
bestHeading /= gBestHeadingInfluence + gLastBestHeadingInfluence

gLastBestHeading = bestHeading

# Don't try to turn more than 30 in one go
bestHeading = hMath.CLIP(bestHeading, 30)

if Debug.dodgyDebug:
    if VisionLink.ObstacleGPSValid():
        print "dodgyDog: heading ", thisBestHeading, "->", \
            bestHeading, " (GPS)"
    else:
        print "dodgyDog: heading ", thisBestHeading, "->", \
            bestHeading, " (Local)"

    if abs(bestHeading - Global.ballH) < 10:
        Indicator.showFacePattern([0,2,2,2,0])
    elif bestHeading < Global.ballH - 60:
        Indicator.showFacePattern([3,0,0,0,0])
    elif bestHeading < Global.ballH:
        Indicator.showFacePattern([3,2,0,0,0])
    elif bestHeading > Global.ballH + 60:
        Indicator.showFacePattern([0,0,0,0,3])
    elif bestHeading > Global.ballH:
        Indicator.showFacePattern([0,0,0,2,3])

# Wag tail in direction of dodge.
tailH = Indicator.TAIL_H_CENTRED
if bestHeading > localHead + 10:
    tailH = Indicator.TAIL_LEFT * 3/4
elif bestHeading < localHead - 10:
    tailH = Indicator.TAIL_RIGHT * 3/4

Indicator.finalValues[Indicator.TailH] = tailH
#Indicator.finalValues[Indicator.TailV] = tailV

fwd = Action.MAX_FORWARD
turnccw = bestHeading

```

```

closeDist = CLOSE_DODGY_OBSTACLE_DIST
if closeDist > distToDest - MIN_DODGY_TARGET_DIST:
    closeDist = distToDest - MIN_DODGY_TARGET_DIST
closePoint = hMath.polarToCart(closeDist, localHead)
closeObsIntensity = VisionLink.getNoObstacleBetween(0, 0,
    int(closePoint[0]), int(closePoint[1]),
    CLOSE_CORRIDOR_WIDTH, gUseMinDist,
    gUseMinIntensity, Constant.OBS_USE_NONE)

if closeObsIntensity > MIN_OBS_IN_CORRIDOR:
    if bestHeading > localHead:
        gLeft = Action.MAX_LEFT
    else:
        gLeft = -Action.MAX_LEFT

    #turnccw /= 2.0
    gSidestepCounter = SIDESTEP_FOR
else:
    if gSidestepCounter > 0:
        gSidestepCounter -= 1
    else:
        gLeft = 0

if Debug.dodgyDebug and gLeft != 0:
    if bestHeading > localHead:
        print "sDodgyDog: Sidestepping LEFT (" , gSidestepCounter, ")"
    else:
        print "sDodgyDog: Sidestepping RIGHT (" , gSidestepCounter, ")"

Action.walk(fwd, gLeft, turnccw, minorWalkType=Action.SkeFastForwardMWT)

return Constant.STATE_EXECUTING

else:
    if Debug.dodgyDebug:
        print "getBestGap failed - can't be dodgy"
    return Constant.STATE_FAILED

```

Programming Code for getDogShifts (program to calculate linear shift table)

Written in Java

/*

Copyright 2004 The University of New South Wales (UNSW) and National ICT Australia (NICTA).

This file is part of the 2004 team rUNSWift RoboCup entry. You may redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version as modified below. As the original licensors, we add the following conditions to that license:

In paragraph 2.b), the phrase "distribute or publish" should be interpreted to include entry into a competition, and hence the source of any derived work entered into a competition must be made available to all parties involved in that competition under the terms of this license.

In addition, if the authors of a derived work publish any conference proceedings, journal articles or other academic papers describing that derived work, then appropriate academic citations to the original work must be included in that publication.

This rUNSWift source is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this source code; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

*/

```
import java.io.*;
import java.util.*;
import java.lang.*;
```

```
public class getDogShifts
{
    static final int WIDTH = 208;
    static final int HEIGHT = 159;
```

```

static final int COLOURS = 7;
static final int COLOUR_ELEMENTS = 3;
static final int DOGS = 12;
static final int SAMPLES = 1000;

static final int HALF_HEIGHT = (int)Math.floor(HEIGHT / 2);
static final int HALF_WIDTH = (int)Math.floor(WIDTH / 2);

static final int SAMPLE_HEIGHT_FROM = HALF_HEIGHT - 5;
static final int SAMPLE_HEIGHT_TO = HALF_HEIGHT + 5;
static final int SAMPLE_WIDTH_FROM = HALF_WIDTH - 5;
static final int SAMPLE_WIDTH_TO = HALF_WIDTH + 5;

protected static byte y[] = new byte[WIDTH];
protected static byte u[] = new byte[WIDTH];
protected static byte v[] = new byte[WIDTH];
protected static byte rest[] = new byte[4*WIDTH];

public static String usage =
"Usage: java getDogShifts <image_directory> [base_dog]\n"
+"Help: java getDogShifts --help";

public static String help =
"\ngetDogShifts\n"
+"-----\n"
+"this program will process a group of calibration images\n"
+"and for each Y, U and V of every dog it will return the function\n"
+"to transform the value so that it matches that of the base dog's.\n"
+"n"
+"Step 1)\n"
+"for each of the 8 dogs take 10 pictures in BFL format (11/3/05) of\n"
+"EACH of the following 7 colours (i.e. 560 pictures all up);\n"
+"yellow, green, blue, red, lightblue, white, pink\n"
+"the pictures should be taken so the colour fills roughly the entire screen\n"
+"each picture should be named as follows;\n"
+"<first letter of dog colour><jersey number>-<colour>-<abstract identifier>.BFL\n"
+"for example r2-yellow-234.BFL , b3-lightblue-YUV32.BFL\n"
+"IMPORTANT NOTE: these photos should be taken running code that does not\n"
+"already shift the values, i.e. comment out all non default entries in\n"
+"DogMacLookupType in VisualCortex.cc, recompile the dog and use that code\n"
+"to take images\n"
+"Note: you may now optionally include four white dogs\n"
+"whose image file names should start with w1, w2, w3 & w4\n"
+"n"

```

```

+"Step 2)\n"
+"put all the pictures in a directory\n"
+"run getDogShifts on that directory, note you can specify\n"
+"a base dog in the form <first letter of dog colour><jersey number>\n"
+"however b1 will be taken as default if you do not\n"
+"\n"
+"Step 3)\n"
+"in that directory there should now be a group of .csv files, these are designed\n"
+"to be opened in OpenOffice which should have no trouble importing them to calc\n"
+"you should set calc's default decimal places to 4 for more accurate results\n"
+"for each Y, U and V of each of the 7 non base dogs there is a function of the\n"
+"form  $y = m * x + b$  that maps the value to that of the base dog, in the .csv files\n"
+"the value of 'm' is given by the '1/m' field and the value of 'b' is given by\n"
+"the '-b/m' field\n"
+"\n"
+"Step 4)\n"
+"input these values into the VisualCortex.cc file and make sure the current\n"
+"colour calibration is for the base dog\n";

```

```

public static int byte2UInt(byte b)
{
    int i,s;

    s = (b >> 7) & 0x01;    // save the sign bit
    b &= 0x7f;              // strip the sign bit
    i = (((int) b) | (s<<7)); // reassemble number

    return i;
}

public static void main(String args[]){
    if (args.length < 1 || args.length > 2) {
        System.err.println(usage);
        return;
    }

    if (args[0].compareTo("--help") == 0){
        System.err.println(help);
        return;
    }

    int i,j,k,m;
    String path, baseDog;

    path = args[0];

```



```

        if (args.length > 1){baseDog = args[1];}
        else{baseDog = "b1";}

        File dir = new File(path);
        if (!dir.isDirectory()) {
            System.err.println(path+" is not a directory");
            return;
        }

        path += File.separator;

String[] colourArray =
    { "yellow","red","blue","green","pink","lightblue","white"};

HashMap<String, Integer> colours = new HashMap<String, Integer>();
//initialise colour map
for(i = 0; i < COLOURS; i++)
    colours.put(colourArray[i],new Integer(i));

String[] dogArray = { "r1","r2","r3","r4","b1","b2","b3","b4","w1","w2","w3","w4"};
int[] gotDogData = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

String[] dogKeys = new String[DOGS-1];

HashMap<String, Integer> dogs = new HashMap<String, Integer>();
int dogi = 0;
for (i = 0; i < DOGS; i++){

    if (baseDog.compareTo(dogArray[i]) != 0){
        dogs.put(dogArray[i], new Integer(dogi));
        dogKeys[dogi] = dogArray[i];
        dogi++;
    }
}

        String[] fileList = dir.list();
String name, colour, dog;
int firstHyphen, secondHyphen;
int t_dog, t_colour, t_sample;
int lastcolour = -1;

float baseDogVals[][] = new float[COLOURS][COLOUR_ELEMENTS];
int baseDogValsCount[] = new int[COLOURS];
int restDogVals[][][][] = new int[DOGS-1][COLOURS][COLOUR_ELEMENTS][SAMPLES];

```

```

int sampleIndex[][] = new int[DOGS-1][COLOURS];

//initialise baseDogValsCount array to zero
for (i = 0; i < COLOURS; i++)
    baseDogValsCount[i] = 0;

//initialise restDogVals to -1
//so we can detect if less than SAMPLE values are taken
for (i = 0; i < (DOGS-1); i++)
    for (j = 0; j < COLOURS; j++)
        for (k = 0; k < COLOUR_ELEMENTS; k++)
            for (m = 0; m < SAMPLES; m++)
                restDogVals[i][j][k][m] = -1;

//initialise sampleIndex array to zero
for (i = 0; i < (DOGS-1); i++)
    for (j = 0; j < COLOURS; j++)
        sampleIndex[i][j] = 0;

    try {
        FileInputStream in = null;

        for(k = 0; k < fileList.length; k++) {
            if(!fileList[k].toLowerCase().endsWith(".bfl"))
                continue;

            firstHyphen = fileList[k].indexOf('-');
            secondHyphen = fileList[k].indexOf('-', firstHyphen+1);
            colour = fileList[k].substring(firstHyphen + 1, secondHyphen);
            dog = fileList[k].substring(0,firstHyphen);

            name = path+fileList[k];
            in = new FileInputStream(name);
            if (lastcolour != colours.get(colour)){
                System.out.println();
                System.out.print("Reading "+dog+" "+colour);
                lastcolour = colours.get(colour);
            }
            else System.out.print(".");

            for (i = 0; i < SAMPLE_HEIGHT_TO; i++) {

                in.read(y);
                in.read(u);
                in.read(v);
                in.read(rest);
            }
        }
    }

```

```

//discard data until we come to sample area
    if (i < SAMPLE_HEIGHT_FROM) continue;

    for (j = SAMPLE_WIDTH_FROM; j < SAMPLE_WIDTH_TO;
j++) {

        t_colour = (colours.get(colour)).intValue();

//if file is data for the base dog
        if (baseDog.compareTo(dog) == 0)
        {
            baseDogVals[t_colour][0] =
baseDogVals[t_colour][0] + byte2UInt(y[j]);

baseDogVals[t_colour][1] =
baseDogVals[t_colour][1] + byte2UInt(u[j]);

            baseDogVals[t_colour][2] =
baseDogVals[t_colour][2] + byte2UInt(v[j]);

            baseDogValsCount[t_colour] =
baseDogValsCount[t_colour] + 1;

        }
        else
        {

            t_dog = (dogs.get(dog)).intValue();
gotDogData[t_dog] = 1;

            t_sample = sampleIndex[t_dog][t_colour];

if (t_sample < SAMPLES){

restDogVals[t_dog][t_colour][0][t_sample] =
byte2UInt(y[j]);

            restDogVals[t_dog][t_colour][1][t_sample] =
byte2UInt(u[j]);

            restDogVals[t_dog][t_colour][2][t_sample] =
byte2UInt(v[j]);
}
}

```

```

        //increment sample index
        sampleIndex[t_dog][t_colour] =
            sampleIndex[t_dog][t_colour] + 1;
    }

}

}

in.close();
}

//calculate the average Y,U and V for each
//colour for the base dog
for (i = 0; i < COLOURS; i++){

    for (j = 0; j < COLOUR_ELEMENTS; j++){
        baseDogVals[i][j] =
            baseDogVals[i][j] / baseDogValsCount[i];
    }
}

PrintWriter out = null;
System.out.println();

String[] LineEst = { "LINEST(A14:A7013;B14:B7013;1;1)",
    "LINEST(D14:D7013;E14:E7013;1;1)",
    "LINEST(G14:G7013;H14:H7013;1;1)"};

String savefile;

for (i = 0; i < (DOGS-1); i++){

    if (gotDogData[i] == 0) {continue;}

    savefile = path+"out-"+dogKeys[i]+"-withbase-"+baseDog+".csv";
    out = new PrintWriter(new FileWriter(savefile),true);
    System.out.println("Writing "+savefile);

    out.println();
    out.print("LINEST FOR Y,,LINEST FOR U,,LINEST FOR V,");

```

```

out.println();

for (j = 1; j < 6; j++){

    out.print("=INDEX("+LineEst[0]+";"+String.valueOf(j)+";1,");
    out.print("=INDEX("+LineEst[0]+";"+String.valueOf(j)+";2,,");

    out.print("=INDEX("+LineEst[1]+";"+String.valueOf(j)+";1,");
    out.print("=INDEX("+LineEst[1]+";"+String.valueOf(j)+";2,,");

    out.print("=INDEX("+LineEst[2]+";"+String.valueOf(j)+";1,");
    out.print("=INDEX("+LineEst[2]+";"+String.valueOf(j)+";2,,");

    out.println();
}

out.println();

out.print("1/m,-b/m,,1/m,-b/m,,1/m,-b/m,,");

out.println();

out.print("=1/A3,=-B3/A3,,=1/D3,=-E3/D3,,=1/G3,=-H3/G3,,");
out.println();

out.println();
out.print("Y,,");
out.print("U,,");
out.print("V,");
out.println();

out.print(dogKeys[i]+","+"baseDog+","");
out.print(dogKeys[i]+","+"baseDog+","");
out.print(dogKeys[i]+","+"baseDog+","");
out.println();

for(j = 0; j < COLOURS; j++){

    for (k = 0; k < SAMPLES; k++){

        if (restDogVals[i][j][0][k] == -1){
            out.print(",,LIMITED SAMPLES,");
            out.print(",,LIMITED SAMPLES,");
            out.print(",,LIMITED SAMPLES,");
        }
    }
}

```

```

else
{
    for (m = 0; m < COLOUR_ELEMENTS; m++){
        out.print(String.valueOf(restDogVals[i][j][m][k])+",");
        out.print(String.valueOf(baseDogVals[j][m])+",");
        out.print(colourArray[j]+",");
    }
}
out.println();

}

}

out.println();
out.print("Analysis");
out.println();

String a;
String b;
String base;
String avg;
String lin;

out.println();
out.print("Y,base val,dog avg,*diff*,lin shift,*diff*,linD < avgD");
out.println();

for(j = 0; j < COLOURS; j++){

    a = String.valueOf((j*1000)+14);
    b = String.valueOf(((j+1)*1000)+13);
    base = "B"+a;
    avg = "AVERAGE(A"+a+":A"+b+")";
    lin = "((A"+b+"*A10)+B10)";

    out.print(colourArray[j]+",");
    out.print("=" + base + ",");
    out.print("=" + avg + ",");
    out.print("=ABS((" + base + ")-(" + avg + ")),");
    out.print("=" + lin + ",");
    out.print("=ABS((" + base + ")-(" + lin + ")),");

    a = String.valueOf((COLOURS*1000)+18+j);
    out.print("=F"+a+"<D"+a);

```

```

    out.println();
}

out.println();
out.println();

out.println();
out.print("U,base val,dog avg,*diff*,lin shift,*diff*,linD < avgD");
out.println();

for(j = 0; j < COLOURS; j++){

    a = String.valueOf((j*1000)+14);
    b = String.valueOf(((j+1)*1000)+13);
    base = "E"+a;
    avg = "AVERAGE(D"+a+":D"+b+"");
    lin = "((D"+b+"*D10)+E10)";

    out.print(colourArray[j]+",");
    out.print("=" + base + ",");
    out.print("=" + avg + ",");
    out.print("=ABS((" + base + ")-(" + avg + ")),");
    out.print("=" + lin + ",");
    out.print("=ABS((" + base + ")-(" + lin + ")),");

    a = String.valueOf((COLOURS*1000)+29+j);
    out.print("=F"+a+"<D"+a);

    out.println();
}

out.println();
out.println();

out.println();
out.print("V,base val,dog avg,*diff*,lin shift,*diff*,linD < avgD");
out.println();

for(j = 0; j < COLOURS; j++){

    a = String.valueOf((j*1000)+14);
    b = String.valueOf(((j+1)*1000)+13);
    base = "H"+a;
    avg = "AVERAGE(G"+a+":G"+b+"");
    lin = "((G"+b+"*G10)+H10)";

```

```

        out.print(colourArray[j]+",");
        out.print("=" + base + ",");
        out.print("=" + avg + ",");
        out.print("=ABS((" + base + ")-( " + avg + ")),");
        out.print("=" + lin + ",");
        out.print("=ABS((" + base + ")-( " + lin + ")),");

        a = String.valueOf((COLOURS*1000)+40+j);
        out.print("=F"+a+"<D"+a);

        out.println();
    }

    out.println();

    out.print(",,Total Diff,,Total Diff");

    out.println();

    out.print(",,=SUM(D7018:D7046),=SUM(F7018:F7046),=SUM(G7018:G7046),");
    out.print("of 21");

    out.println();

    out.close();
}

        } catch (Exception e) {
            System.out.println(e);
            e.printStackTrace();
        }
    }
}

```


Programming Code for reconfigureShifts (program to reconfigure linear shift table)

Written in C++

/*

Copyright 2005 The University of New South Wales (UNSW) and National ICT Australia (NICTA).

This file is part of the 2005 team rUNSWift RoboCup entry. You may redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version as modified below. As the original licensors, we add the following conditions to that license:

In paragraph 2.b), the phrase "distribute or publish" should be interpreted to include entry into a competition, and hence the source of any derived work entered into a competition must be made available to all parties involved in that competition under the terms of this license.

In addition, if the authors of a derived work publish any conference proceedings, journal articles or other academic papers describing that derived work, then appropriate academic citations to the original work should be included in that publication.

This rUNSWift source is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this source code; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

*/

/* reconfigure linear shift table in LinearShift.h

* to have a different base dog

*/

#include <stdlib.h>

#include "../..../robot/vision/LinearShifts.h"

```

using namespace std;

void usage(void);

int main(int argc, char* argv[]) {

    bool oldDogKnown = true;

    // check command line arguments for dog ips
    if (argc < 2) {
        usage();
        exit(1);
    }

    if (argc < 3) {
        oldDogKnown = false;
    }
    else if (argv[1] == argv[2])
    {
        fprintf(stderr, "ERROR: old base dog cannot be the same as new base dog\n");
        exit(1);
    }

    int i=0;
    int numFields = sizeof(DogMacLookup)/sizeof(DogMacLookupType);

    int newBaseDog = 0;
    bool oldBaseDogFound = false;
    bool newBaseDogFound = false;

    for (i=0; i < numFields; i++) {

        if(DogMacLookup[i].mac == strtoul(argv[1], NULL, 16)) {
            newBaseDog = i;
            newBaseDogFound = true;
            continue;
        }

        if ((oldDogKnown) && (DogMacLookup[i].mac == strtoul(argv[2], NULL, 16))) {
            oldBaseDogFound = true;
        }
    }

    if (not(newBaseDogFound)) {
        fprintf(stderr, "ERROR: new base dog not found in DogMacLookup table\n");
        exit(1);
    }
}

```

```

}

if (oldBaseDogFound) {
    fprintf(stderr, "ERROR: old base dog is a dog in the lookup table! (it shouldn't be)\n");
    exit(1);
}

DogColourDistortion newBaseDogShift = DogMacLookup[newBaseDog].colMunge;

//calculate new dog shifts
for (i=1; i < numFields; i++) {

    if (i == newBaseDog) continue;

    DogMacLookup[i].colMunge.My = DogMacLookup[i].colMunge.My / newBaseDogShift.My;
    DogMacLookup[i].colMunge.Cy = (DogMacLookup[i].colMunge.Cy - newBaseDogShift.Cy) /
newBaseDogShift.My;

    DogMacLookup[i].colMunge.Mu = DogMacLookup[i].colMunge.Mu / newBaseDogShift.Mu;
    DogMacLookup[i].colMunge.Cu = (DogMacLookup[i].colMunge.Cu - newBaseDogShift.Cu) /
newBaseDogShift.Mu;

    DogMacLookup[i].colMunge.Mv = DogMacLookup[i].colMunge.Mv / newBaseDogShift.Mv;
    DogMacLookup[i].colMunge.Cv = (DogMacLookup[i].colMunge.Cv - newBaseDogShift.Cv) /
newBaseDogShift.Mv;

}

//calculate linear shift for old base dog
if (oldDogKnown) {

    DogMacLookup[0].colMunge.Cy = -DogMacLookup[newBaseDog].colMunge.Cy
        / DogMacLookup[newBaseDog].colMunge.My;
    DogMacLookup[0].colMunge.My = 1 / DogMacLookup[newBaseDog].colMunge.My;

    DogMacLookup[0].colMunge.Cu = -DogMacLookup[newBaseDog].colMunge.Cu
        / DogMacLookup[newBaseDog].colMunge.Mu;
    DogMacLookup[0].colMunge.Mu = 1 / DogMacLookup[newBaseDog].colMunge.Mu;

    DogMacLookup[0].colMunge.Cv = -DogMacLookup[newBaseDog].colMunge.Cv
        / DogMacLookup[newBaseDog].colMunge.Mv;
    DogMacLookup[0].colMunge.Mv = 1 / DogMacLookup[newBaseDog].colMunge.Mv;

    DogMacLookup[0].mac = strtoul(argv[2], NULL, 16);
}

```

```

//assign static entry for new base dog
DogMacLookup[newBaseDog].colMunge.My = 1.0;
DogMacLookup[newBaseDog].colMunge.Cy = 0.0;
DogMacLookup[newBaseDog].colMunge.Mu = 1.0;
DogMacLookup[newBaseDog].colMunge.Cu = 0.0;
DogMacLookup[newBaseDog].colMunge.Mv = 1.0;
DogMacLookup[newBaseDog].colMunge.Cv = 0.0;
DogMacLookup[newBaseDog].mac = 0;

//swap new base dog with first entry
DogMacLookupType tmp;
tmp = DogMacLookup[0];
DogMacLookup[0] = DogMacLookup[newBaseDog];
DogMacLookup[newBaseDog] = tmp;

//print out the new lookup table
bool newDogIsLast = false;
if (newBaseDog == (numFields - 1)) {
    newDogIsLast = true;
}

cout << "DogMacLookupType DogMacLookup[] = {" << endl;
cout << "    // Default Entry (Base Dog = " << argv[1] << ")" << endl;
cout << "    {0, {1.0, 0.0, 1.0, 0.0, 1.0, 0.0}} , " << endl;
for (i=1; i < numFields; i++) {
    if (not(oldDogKnown) && i == newBaseDog) {
        continue;
    }

    cout << "    {";

    printf("0x%x", (unsigned int) DogMacLookup[i].mac );
    cout << ", {" << DogMacLookup[i].colMunge.My << ", "
    << DogMacLookup[i].colMunge.Cy << ", "
    << DogMacLookup[i].colMunge.Mu << ", "
    << DogMacLookup[i].colMunge.Cu << ", "
    << DogMacLookup[i].colMunge.Mv << ", "
    << DogMacLookup[i].colMunge.Cv << "}}";

    if ((i != numFields - 1)
        && not(newDogIsLast) && (not(oldDogKnown)) && (i == (numFields-2)))) {
        cout << ", ";
    }

    cout << endl;
}

```

```

    cout << "};" << endl;

    return 0;
}

void usage(void) {
    char *msg =
        "Usage: reconfigureShifts newBaseDog [oldBaseDog]\n"
        "where;\n"
        "newBaseDog is the mac address of the dog in the table that you want to make the base dog\n"
        "oldBaseDog is the mac address of the base dog currently used in the table\n"
        "the current base dog mac address can be found as a comment at the top\n"
        "of the DogMacLookup table, if the base dog mac address is specified an entry will be\n"
        "created in the table for the shift of that dog, else if it is not known it can\n"
        "be left out and that dog will just use the default linear shift (i.e. none)\n"
        "\n"
        "IMPORTANT: the linear shift table is read in at COMPILE time, so it is vital that you\n"
        "remake this program every time you use it."
        "\n"
        "NOTE: this program does not actually replace the table in LinearShifts.h\n"
        "it prints the new table to stdout so you will need to copy and paste it\n"
        "before it can take effect\n"
        "\n"
        "examples:\n"
        "  reconfigureShifts 0xC6245478 0xc6245481\n"
        "  reconfigureShifts 0xc66b055c 0xc624544b\n";

    fprintf(stderr, msg);
}

```