THE UNIVERSITY OF
NEW SOUTH WALES

rUNSWift
UNSW RoboCup2002 Sony Legged League Team

Zheng (David) Wang, 2253591
James Wong, 2253214
Timothy Tam, 2249914
Benjamin Leung, 2250480
Min Sub Kim, 2252751
James Brooks, 2252495
Albert Chang, 2251971
Nik Von Huben, 2252536

November 5, 2002

Contents

Chapter 1

Introduction

Section author: Timothy Tam and Zheng David Wang

1.1 Abstract

In this report we present a comprehensive description of the software system developed to compete in the Sony legged robot league competition at RoboCup 2002 taken place in Fukuoka, Japan. We won the RoboCup 2002 Challenge and were placed second in the main tournament. We describe a system that controls a quadruped robot (Sony AIBO ERS-210), allowing a team of these robots to engage in competitive games of soccer against other teams from a wide range of institutions around the world. This report describes the individual software sub-systems and the overall agent software architecture developed for the competition.

1.2 Purpose

The purpose of the RoboCup competition is to stimulate research in advanced mobile robotics. For a mobile robot to operate effectively it must be capable of at least the following:

- perceive its environment through its sensors
- use its sensory inputs to identify objects in the environment
- use its sensors to locate the robot relative to other objects in the environment
- plan a set of actions in order to achieve some goal

- execute the actions, monitoring the progress of the robot with respect to its goal

The robot's planning becomes much more complex when its environment includes other active agents, such as other robots. These may cooperate with the robot or try to thwart its attempts to achieve its goals.

A competitive game like soccer requires all of the above skills. Thus, RoboCup was created to provide an incentive for researchers to work on their individual problems and integrate their solutions in a very concrete task. The benefits of the competition are evident in the amount of progress made from one year to the next. The competitive nature of the research has forced participants to critically evaluate their methods in a realistic task as opposed to experiments in a controlled laboratory. It has also forced researchers to face practical hardware and real time constraints.

1.3 Background

The Sony legged robot league is one of five leagues of the Robocup competition. UNSW has participated since 1999 and has an excellent achievement record to date; runners up in 1999 and champions in 2000 and 2001. UNSW again put up a strong performance in the competition this year finishing runner up. The final against CMU was extremely close, the final score being 3-3 and UNSW losing the penalty shootout. Each game consists of 10 minutes halves with 4 robots on each side (one goalie and three players), the objective of the game being to score as many goals as possible. Beacons are placed at the edge of the field and all objects within the field are colour coded, to aid the robots in determining their location.

The robots used in the legged league are ERS-210 model Sony entertainment robots. However different in contrast to the other leagues physical modifications to the robots is disallowed, thus the development relies entirely on software programming.

1.4 Robot Description Reference

The ERS-210 model is the same robot used in the 2001 competition. The main components of the robot include a camera in its snout where a camera image is captured on average 25 times a second, a memory stick slot to load the executable binary and four legs, each with three degrees of freedom in movement. For a more detailed description of the ERS-210 please refer to the 2001 report.

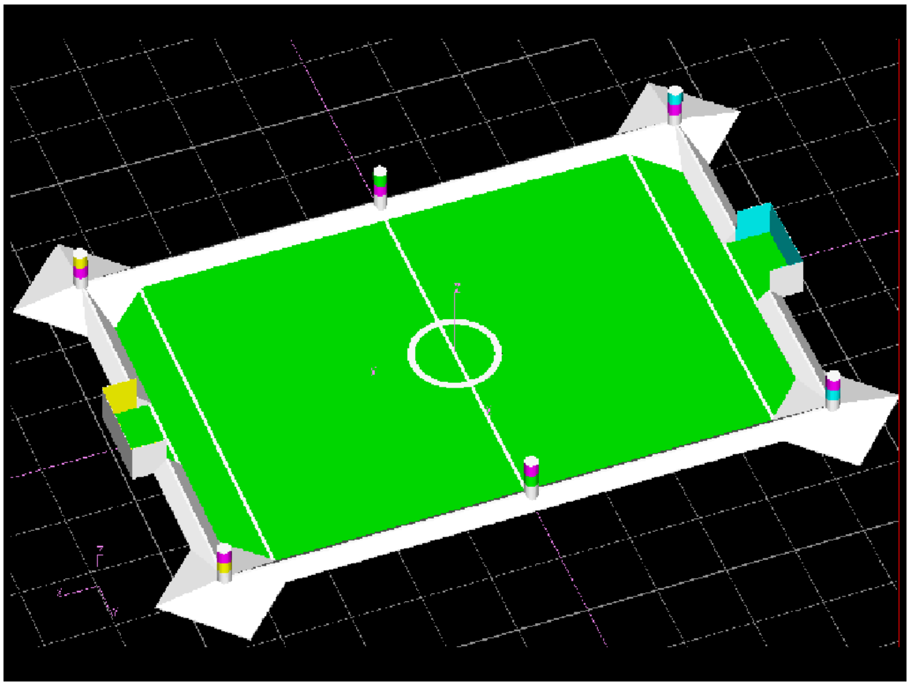


Figure 1.1: RoboCup Field 2002

1.5 Changes from Previous Year Competition

1.5.1 Field change

This field size this year was enlarged to 2.9m x 4.4m compared to 2.0m x 3.0m in the 2001 competition. The beacon width increased by 1cm where as the goal size remained unchanged. All objects were placed in the same position proportionally in comparison to the old field. The increase in field size posed greater challenges in vision and localisation and existing legacy routines from 2001 had to be modified in order cope with the new field.

1.5.2 Number of Robots

Due to the increased field size the number of robots on each side was increased from three to four. An additional robot inspired new cooporation strategies and positioning play.

1.5.3 Wireless

In the 2002 competition each robot had a wireless card installed into it. This allowed for robot to robot communciation via ethernet during gameplay and also greater control of the robots by the referees. Enormous effort was done on developing wireless in 2002. However the bulk of the effort was spent on the building of debugging tools which aided the team during development. For example a tool was developed to display a robot's perception of the objects on the field in real time. This was achieved by sending the information via a wireless link to a computer with debugging software installed.

1.5.4 Changes in Game Rules

Placement of Robot Positions during Kickoff

The attacking team places the robots first. Two of these robots are allowed to be placed anywhere between the center line and the middle of their half. The remaining robots on both teams (except the goalies) must be placed anywhere between the middle of their half and their own penalty line.

Illegal Defence

In 2001 if a robot that is not the goal keeper enters its own penalty area it is removed immediately and replaced on the halfway line heading for the centre circle. In 2002 this rule still holds however if the same robot commits the offence three times then it is sent off for thirty seconds until the next game restart. A game restart occurs when a goal is scored or "game stuck" is declared by the referee.

Goalie Charging

No robot is allowed to make physical contact with an opponent goalie. If such a situation occurs the offending robot is taken off the field for 60 seconds (compared to thirty seconds in the previous year's competition) then replaced at the halfway line heading for the centre circle.

1.6 Architecture

Software modularisation enables ease of independent development. It allows code organisation and simplifies the debugging process by localising the error. In essence the team can develop modules independently and then glue them together to complete the whole system.

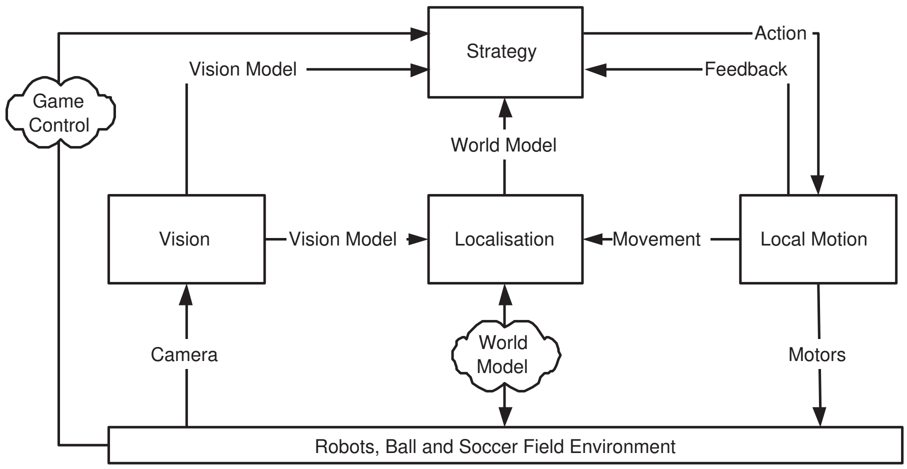


Figure 1.2: Architecture of the UNSW2002 rUNSWift software

The UNSW2002 rUNSWift software is divided into four main modules. Each module receives information from the environment or other modules. Some modules pass information onto or receives data from the wireless network. Information transferred over the wireless network is represented by a cloud in the diagram in figure 1.2. The details of each module will be described in the following sections.

1.7 Architecture Overview

1.7.1 Vision

All the information the robot perceives about the environment initially comes from the vision system. The objects in the RoboCup field are colour coded and are one of nine distinct colours. The goal of the vision module is to be able to accurately recognise objects on the field. This information is then passed onto the other modules for decision making.

The first stage of the vision system is to teach the robot to recognise certain colours. A nearest neighbour learning algorithm was used this year to replace the previous year's decision tree algorithm. The newer learning algorithm outperforms the older one as it recognises certains colours better and the learning time is significantly less. After learning, a colour lookup is produced which maps every YUV coordinate to one of the nine distinct colours. The robot uses this lookup table to convert every pixel in its camera image to a colour it can recognise.

The second stage of the vision system is to take the converted camera image and form "blobs" of the same colour. These blobs are mapped to objects the robot knows exists in the field based on the size of the blob and the position of the blob relative to other blobs. Sanity checks are included to ensure that objects are accurately recognised for example a goal cannot appear on top of a beacon.

1.7.2 Localisation

Localisation is the process of finding where you are on the field and finding the location of other objects. The resultant information is modelled in 2 forms: World Model and Wireless Model. The world model is derived by combining the information from the robot's vision system and odometry from the locomotion module. It holds the location and heading of the robot itself, the ball and other robots. The wireless model is derived by combining the robot's own world model and the team-mates' world models. It holds the location and heading of the ball, the robots and the team-mates belief of their location.

When a robot localises itselfm it will triangulate its location if the heading and distance of 2 beacons available. If only a single beacons or a goals is seen, the robot moves its location in the direction of the object.

The Kalman Filter is applied this year to combine the new information from vision with the previous localisation information. It uses statistically collected data for updating the information. This reflects the true uncertainty of localisation.

1.7.3 Locomotion

The locomotion module is responsible for handling the robot motion. For the purposes of playing soccer, this is divided largely into two categories: walking and kicking.

The walking motions are driven via a parameter set that describes the dimensions of the walk. These dimensions include values such as the length of each stride, the height of each step, and the duration of each step. The kicking motions are mostly hard-coded routines that simply execute when called.

The locomotion module spreads over the classes EffectorCommander, PWalk, and ActionPro. EffectorCommander operates at the lowest level, sending the values to the effectors. PWalk contains the actual calculations used to execute the walking and kicking routines, handling the leg motion. ActionPro operates just above PWalk, and handles most of the head motion.

1.7.4 Strategy

With the introduction of a fourth robot into the team in this years competition, the high level strategy was totally rewritten, only basic routines such as finding the ball and moving to a particular (x, y) coordinate on the field were retained. The addition of wireless communication between the robots allowed for better cooporation and positioning among team mates. Improvements in skills such as the robot spinning with the ball trapped under its snout, a stronger turn kick and a very fast kick allowed for vast improvements in the ball control and the choice of kicks when the robot has contained control of the ball.

Compared the previous years the overall strategy is still very aggressive. It was found from playing practise matches that simple strategy where robots aggressively go for the ball beats complex strategy where robots try to do complex positioning or cooporative play. Thus the result strategy has robots constantly looking for the ball and moving towards it.

There are four main components of the high level strategy. The robot firstly checks to see if it has lost the ball and if it has it goes into the find ball routine. It then checks to see whether another team mate robot is closer to the ball and if so the robot tries to move into an optimal position. Note that this check is only performed when the robot itself is reasonably close to the ball to maintain aggressiveness. If the robot does not move into an optimal position it approaches the ball in a strategic way to trap the ball under its chin before choosing a kick to execute.

Chapter 2

Vision Component

Section author: James Wong and Timothy Tam

2.1 Introduction/Goals/Overview

The RoboCup field and all objects on it are one of nine distinct colours. The vision module is at the lowest level of all modules, and every other module in the RoboCup code infrastructure is dependent on information provided by the vision system in some way. Hence it is crucial that the vision system is functioning adequately otherwise the robot will behave erratically.

The goal of the vision system is to be able to recognise objects on the field correctly, for example see and distinguish a blue-pink beacon from a pink-blue beacon. Currently this is achieved by feeding training data about the colours of objects into a learning algorithm. The algorithm outputs a colour lookup table that contains the learned colour of each Y U V value. The table is stored in the robot's memory, and on every camera frame the dog translates the camera image into colours it recognises by using the lookup table. Then via blob formation (looking at groups of pixels of the same colour) objects are recognised and bounding boxes are drawn around them.

2.2 Motivation

During RoboCup 2001 there were two vision learning techniques in place: polygon learning and decision tree. Initially polygon learning was used in the early rounds of competition, and then decision tree was used in the finals. It was found that these techniques had a speed bottleneck in the learning process. A small mistake in the calibration process would result in waiting for excessive periods of time while the learning algorithm was re-run. A new vision learning algorithm that ran quickly and allowed for changes to take effect with minimum delay was desired.

Both the polygon learning and decision tree techniques are very crude in the way they deal with the YUV colourspace. The polygon learning technique doesn't look at the distinct 256 Y coordinates but instead compresses the Y axis into 18 planes. The crudeness of the methods results in a loss of detail in the learning of the colours which ultimately reduces the dog's ability to distinguish between colours. A new technique that looked at the training data in more detail was desired.

The two older vision learning techniques both essentially outputted a colour lookup table which mapped any YUV coordinate to one of the nine colours of the field. Although a YUV coordinate can only have one colour in most cases, there are situations where it can have two. For example consider the colours background and dark blue. A YUV coordinate that is somewhere between these two colours has to be classified as either one, lets say it gets classified as background. Using the old techniques, a pixel with this YUV value would get classified as background regardless of its location in the image. However because this YUV coordinate is a borderline case i.e. it can be either blue dog or background it would make sense to classify it as background if the pixel is in a region of the image which is predominantly background and classify it as blue dog if it's in a region of the image which is a part of a blue dog. A vision learning algorithm that takes into account both the YUV coordinate of the pixel and the location of the pixel within the image to determine the colour would improve the vision.

2.3 Learning Algorithm (Nearest Neighbour)

The new vision learning technique is a straight nearest neighbour algorithm. Firstly it goes through the entire YUV colourspace, marking specific coordinates as colours determined from the training pixels. Then the learning process consisted of each unmarked pixel determining its colour by setting it as the colour of the closest marked pixel. If the unmarked pixel is not within a defined threshold of a training pixel then it is set a background colour. This is so that the colours are not over generalised.

The algorithm can be viewed visually as many distinct separated coordinates that are marked to be certain colours (the training data) in the three dimensional YUV colourspace. Coordinates marked the same colour are grouped into three dimensional "blobs" of a certain colour.

This new technique means the colour lookup table would be much larger as we are now storing the entire YUV colourspace. However due to memory restrictions on the ERS-210 the colourspace was compressed into 128x128x128 coordinates. Therefore the lookup table size is 128*128*128*1 = 2.0MB as the colour can be stored as an unsigned char — one byte.

2.4 Calibration Process

2.4.1 Wireless Picture Taking

Every object in the RoboCup field is one of nine colours. To teach the dog how to recognise the specific colours, training samples of colours must be collected. As the colour of the objects greatly depends on factors such as the position of the main light sources and amount of ambient light, the training samples must be recollected every time there is a major change in the lighting conditions.

The training examples are collected in the form of a BFL image and pixels in the image are marked to be one of the nine colours. The camera inside the robot captures a 176 x 144 pixel image. The BFL format stores rows of pixels together and for each row in the captured image there exists one row of Y coordinates, followed by one line of U coordinates, followed by one line of V coordinates, and one line of what the robot classified the original row to be. Thus the first row of a BFL image will contain the 176 Y coordinates of the first 176 pixels in the captured image, followed by the 176 U coordinates of the first 176 pixels, followed by the 176 V coordinates of the first 176 pixels followed finally by the robots classification of the first 176 pixels.

The development of wireless debugging tools allowed the dog to capture a BFL image in an arbitrary position on the field and have it sent over wireless to be saved on a workstation. In a calibration process several of these images are saved, which form the basis of the training data. It is important to note that the samples collected must cover the range of colours of all objects on the field to ensure the dog adequately learns all the colours. The list of standard images taken is attached in the appendix.

2.4.2 Manual Colour Classification

A java program called HighDisplay that displays BFL images was used to mark the pixels in the image to be a particular colour.

For example in the diagram above all the green pixels in the top half of the beacon are marked to be "2". The number 2 represents the green beacon colour. The table below shows what valid colour each number represents.

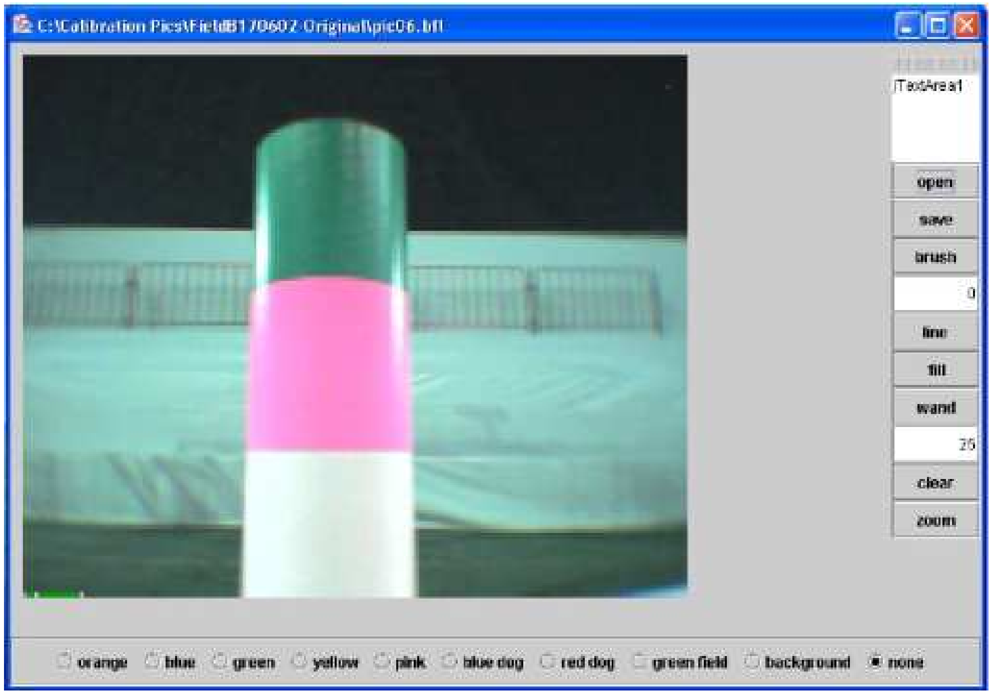


Figure 2.1: Green/pink beacon

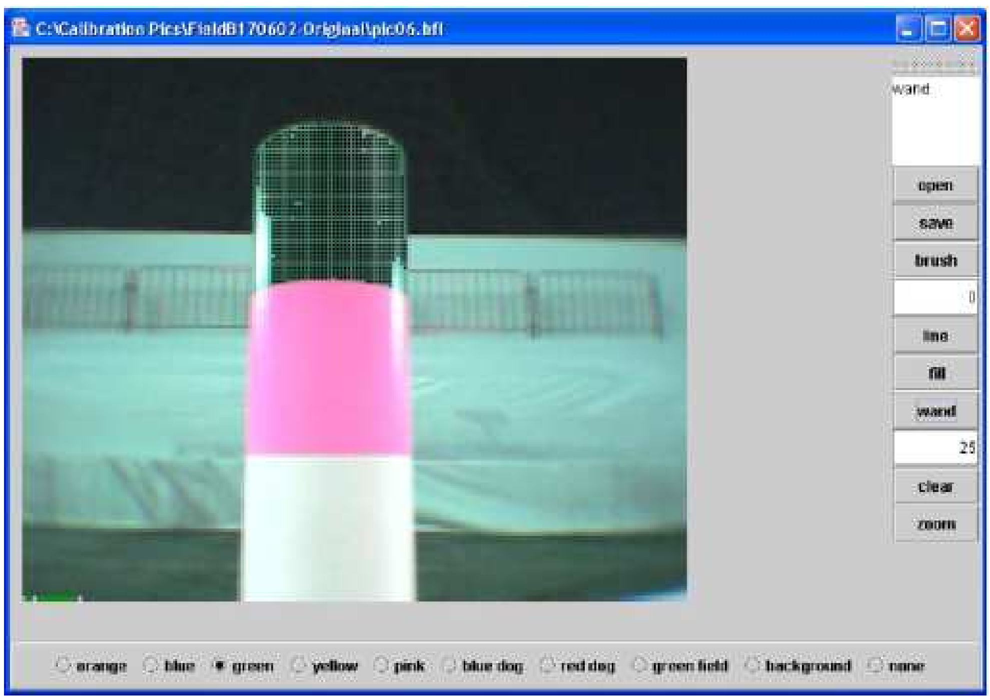


Figure 2.2: Green/pink beacon coloured in

Number	Colour Represented
0	Colour of the ball
1	Colour of the blue half beacon
2	Colour of the green half beacon
3	Colour of the yellow half beacon
4	Colour of the pink half beacon
5	Colour of the blue patches on the dogs
6	Colour of the red patches on the dogs.
7	Colour of the green field
8	Background colour (the remaining colours which the dog does not need to recognise)

In terms of manually colouring in or marking an image we have to be careful about the boundaries of the object. When classifying an image for a ball, we don't want to classify the full image of the ball because the colour of the (orange) ball at the edges are *very* different from the colour of the rest of the ball. If we classify the edge as orange, then we will get a lot more noise in our cplane and increase the likelihood of the robot incorrectly recognising a ball — which has severe consequences in terms of the behaviour of the robot.

At the same time, we do not want to be too conservative because we may not get enough shades of a particular colour. If we do not classify enough pink and blue (the colours of a beacon), then the beacon may appear smaller in area, which will lead to a miscalculation in the localisation routine of the robot.

2.4.3 Generation of training data

The colouring in is performed on every BFL image. Then all the BFL images are converted into large ASCII text file where each line is of the format:

y, u, v, c

where y, u and v are the Y, U, V coordinates of a marked pixel and c is a number from 0-8 representing the marked colour. Note that pixels which were not marked in the BFL image are ignored and a corresponding row is not generated.

The training data is finally generated by assigning each unique y, u, v coordinate to t where t is the colour with the highest tally. For example the following ASCII file BFL image output:

2 50 5 1
5 12 2 5
3 12 4 1
5 12 2 4
5 12 2 4

Generates the following training data output:

2 50 5 1

3 12 4 1
5 12 2 4

The training data is then fed into the nearest neighbour algorithm to generate the colour lookup table.

2.5 Multi-colour

One of the limitations of the vision techniques from the previous years was that each Y U V coordinate could only be assigned one colour. However the colour of a Y U V coordinate is not constant, it is possible that it changes from picture to picture. The multi-colour pixels technique allows a Y U V coordinate to be classified as any combination of the nine colours. It then uses the colour of the nine surrounding pixels to classify the colour of a particular pixel within an image.

From previous years it was found that the colour of the blue dog was consistently classified the worst. This is due to the Y U V coordinates of the colour being close to the Y U V coordinates of background colour. Initially the multi-colour pixels technique was implemented for all nine colours. However it was found that it introduced an unacceptable performance degradation. In the final implementation only multi-colour pixels being labelled as blue dog or background colour were allowed.

Thus in the implementation there exists 10 colours, the first nine are the initial colours, and the 10th colour is one that can be blue dog or background. The multi-colour pixels are generated during the execution of the nearest neighbour algorithm when an unmarked pixel is within the threshold distance of a blue dog training pixel and also within the threshold distance of a background training pixel. Without multi-colour the unmarked pixel would have been labelled as either blue dog or background depending on which training pixel it was closer to.

When a pixel within an image needs to be classified, its colour is determined using the colour lookup table. However if the colour is blue dog or background the colours of the nine surrounding pixels are examined. The number of neighbouring pixels which have the blue dog colour and the number of pixels which have the background colour are determined. Then based on which count has a higher value, the original pixel is marked as that colour. Note that modifications of multicolour pixels have to take place in a copy of the image to prevent bias.

2.6 Compression of Lookup Table

After the colours have been classified, the result is stored in a colour lookup table which will allow the robot to quickly classify all the colours in the image. Given that the memory stick can only hold 16MB and a 256 cube colour lookup table would be 16MB, we need to have a smaller look up table.

The resulting colour lookup table is only 2MB by comparison. It is assumed that the neighbouring pixels in 3D space have the same colour (ie all 8 pixels are classified the same colour. To determine which colour we compress the 8 colour cells to, we take the colour with the highest occurrance out of the 8.

2.7 Colour Cube

The colour cube application is written in C++ using the OpenGL library. The main purpose of the colour cube is to display the YUV values that have been classified. The training samples and the final colour lookup table can be displayed in a three dimensional YUV model. This tool is extremely useful for debugging and determining whether the right camera settings have been set (see "Calibration Verification" section for details).

2.8 Calibration Verification

After the initial colour calibration, steps are taken to verify that the colour calibration has been performed properly, in addition to further improving the accuracy of the colour calibration. The following steps are performed in order.

2.8.1 Re-Check Manual Image Classification

This is a quick check that all the images have been manually classified correctly. All the images are reviewed and checked e.g. a check is performed to see that the ball has not been accidentally classified as "blue dog" for example. Also it is important that that the area outside the object has not been classified as this would result in the blob generated in the nearest neighbour algorithm to expand into incorrect regions.

2.8.2 Examine Colour Cube

The colourcube reads in a binary or ASCII file containing the colours of each of the YUV values. This tool assists greatly in displaying what regions of the YUV space have been classified and what blobs themselves look like. Furthermore the colourcube helps to determine whether the camera setting on the robots are correct. For example, if the camera settings are too dark and the lights on the field are already quite dim, then a clump of blobs would be seen on the edge of the colour cube. Generally the blobs should be *approximately* around the center of the colour cube.

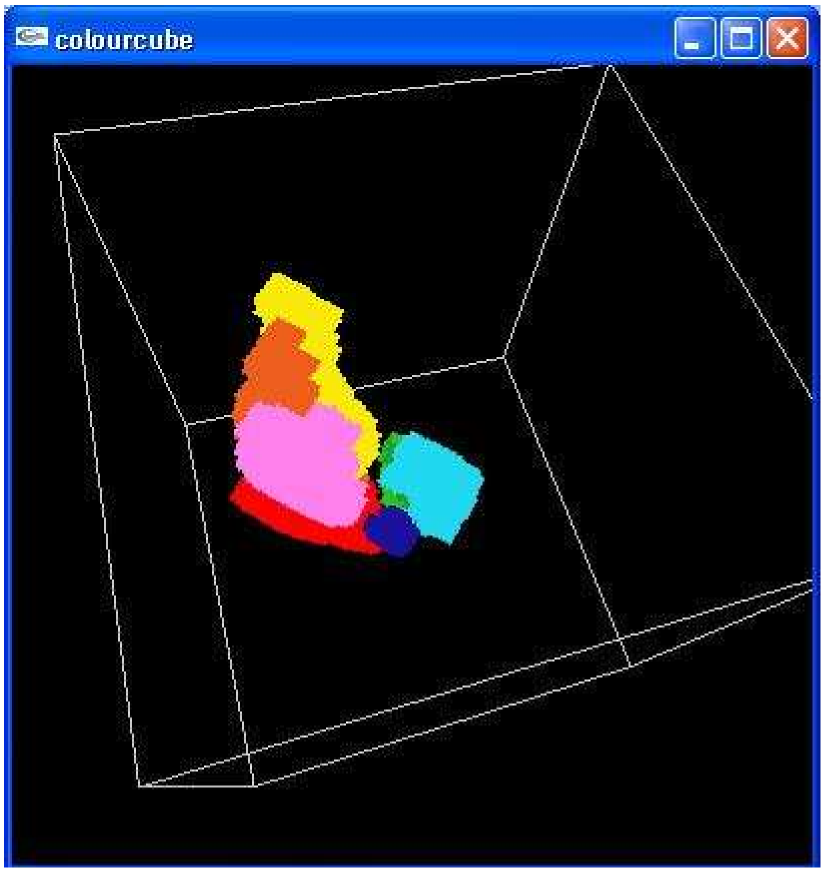


Figure 2.3: Colour Cube

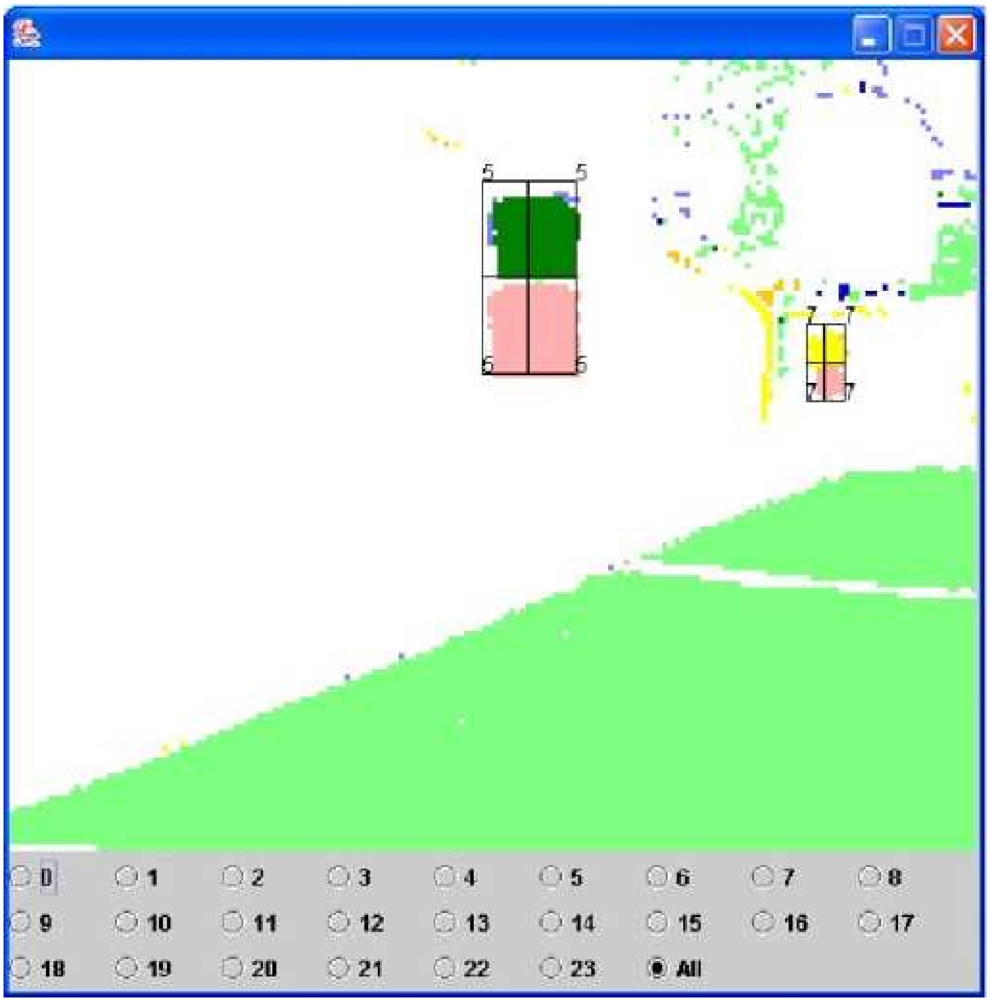


Figure 2.4: Live Cplane Classification of Objects

2.8.3 Examine Live Cplane Feed of Objects

The live cplane is used to determine whether the robot has classified objects correctly. The cplane is also used to determine whether the robot has classified enough of the object. This is vital as the size of the beacons and goals are used in the localisation routine.

All objects in the above image are classified correctly by the robot.

2.8.4 Check Moving Images

Objects appear as a slightly different colour when the robot is moving (the images are more blurred and elongated as well). This is an important consideration due to the fact that the robot is moving (both head and body) quite frequently during game play. While the colour classifier classifies the images correctly most of the time, in some cases, the classifier needs to be fed more training images to be able to correctly classify the colours. To check for moving images, the strategy player called "WirelessControlExercise" is used. This player allows wireless control of the robot through keyboard commands. A check is also performed on the live cplane while the 'normal' players are being run which is arguably the most important test in the determination of the accuracy of the calibration.

2.8.5 Check Extreme Conditions

There are certain tricky situations that are missed during the normal routine check. These extreme conditions have been discovered through observation of the robots for long periods of time and noticing that the robots behave strangely in certain cases. One of the problems encountered was the robots would recognise a ball in the goal when it is moving its head rapidly and close to the goal. Examination of images taken show that the goal is indeed orange.

Another extreme case was discovered when a dog was near the yellow goal. The silver part of the robot's back actually reflects the yellow goal making it appear slightly orange and so the robot recognises a ball on the back of other robots. This only occurs when the robot is extremely close to other robots and these shades of orange are usually calibrated out so that the reflection is classified to be background colour.

2.8.6 Redo the Steps if Needed

To ensure the best possible calibration the above steps are redone to verify that the colour calibration is correct.

2.9 Current Speed Improvements

In practise the whole calibration process took on average one hour from wireless picture taking to calibration verification. The HighDisplay tool already contains efficient painting techniques such as wand and brush so that each pixel does not have to be individually marked.

Furthermore it is rare that a calibration works the first time, problems are usually detected in the verification stage. Thus to correct this, new BFL images need to be taken and coloured in, and the nearest neighbour algorithm is re-run on the new training data. Fortunately the algorithm runs extremely fast, usually taking approximately 10 seconds. This is a substantial speed increase from the vision techniques from the previous years (the decision tree took 45 minutes to

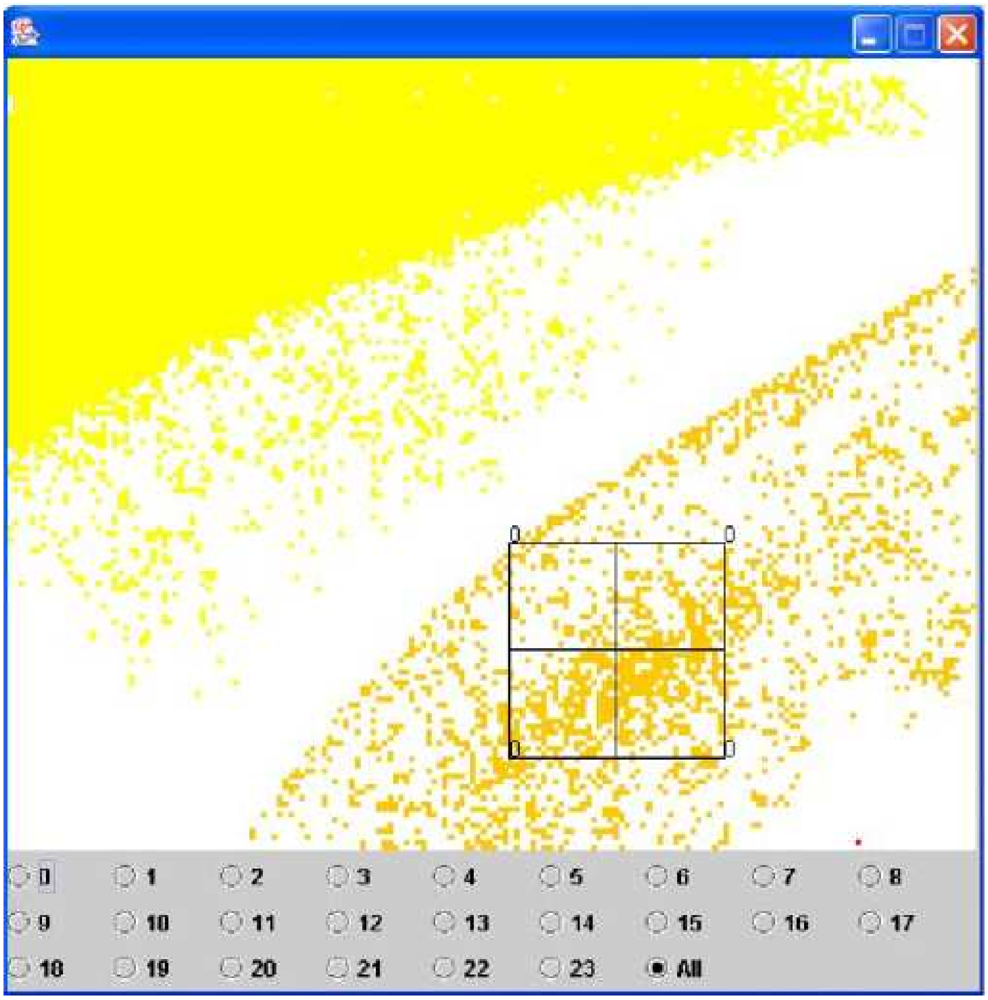


Figure 2.5: Orange in yellow goal

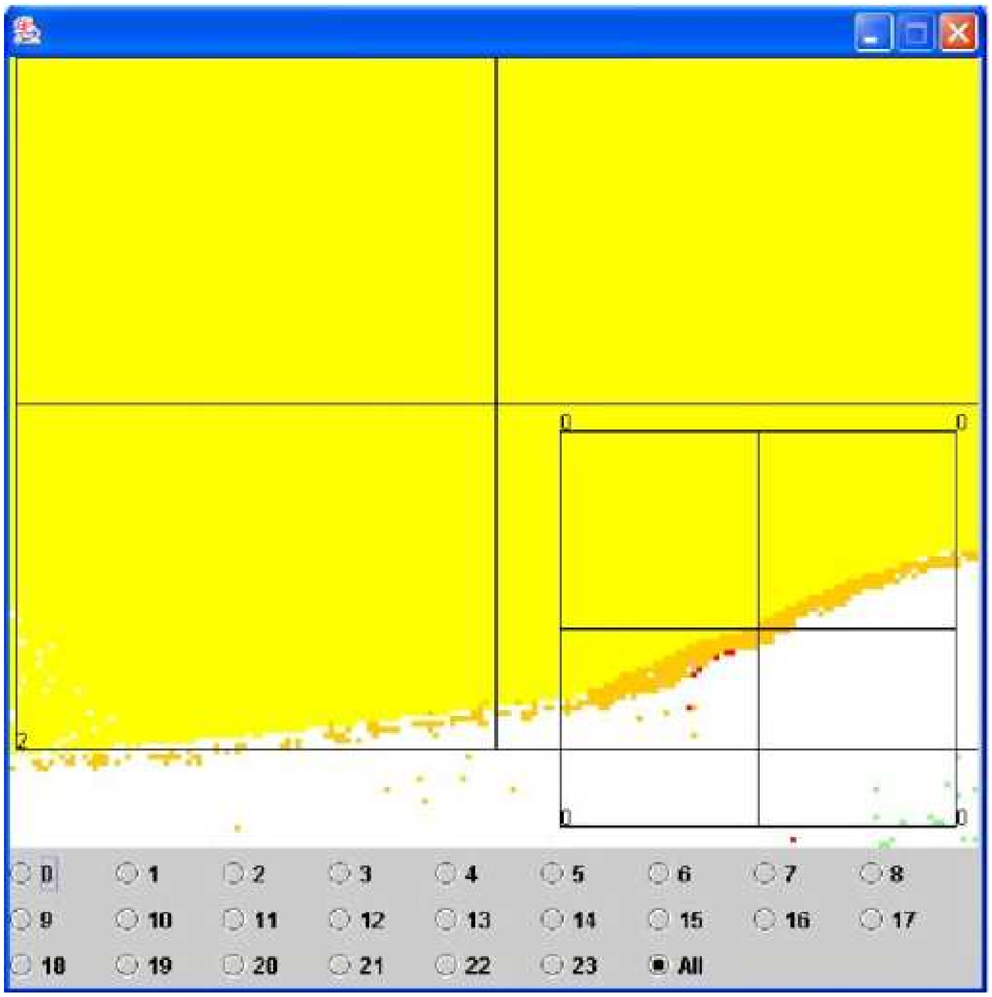


Figure 2.6: Orange reflected off the back of the robot

run) and hence allowed for fine-tuning of the vision to be done with relative ease.

Even if there was a small error in the calibration process pictures would have to be retaken, coloured in and the nearest neighbor algorithm re-run. A technique called dynamic colour calibration was developed where a workstation could modify the colour lookup table of a robot on the fly through wireless communication. A BFL image would be sent from the robot to the workstation and the user could click on any pixel in the BFL image and mark it a particular colour. The Y, U, V coordinate of this pixel and the assigned colour would be sent back to the robot, and the robot would update its colour lookup table to have the Y U V coordinate pointing to the newly assigned colour.

Initially this was thought to be a breakthrough in speed for the calibration process, however through experimentation it was found that this technique had little use except in correcting obviously wrong colours. It was not useful for fine-tuning a calibration at all. As a Y U V coordinate can be different colours in different images, using dynamic colour calibration sometimes fixed an incorrect pixel classification of one image but introduced incorrect classifications in other images. Thus the best results come from taking samples and having a large training data set so that skewed data is minimised which is effectively the method of the current calibration. Furthermore an added disadvantage to this technique is when one modifies the calibration dynamically, a BFL image is not generated for the modification hence there is no training of the data. Thus one cannot fine tune the calibration by taking additional BFL images at a later date and rerunning the algorithm.

Each time verification takes place the new lookup table must be copied onto a memory stick and the dog rebooted. This can be tedious and take up time if done multiple times (which is usually the case in calibration verification) so the ability to send the lookup table over wireless and straight into the dog was developed. This improved the speed of the calibration verification process and once the lookup table was finalised it only need to be copied onto the memory stick once.

2.10 Future Improvements

The bottleneck in the whole calibration process is by far the colouring in of the BFL images. It is possible to do some type of shape recognition to automate the marking of pixels. For example if the BFL image is known to contain a large shot of a green-pink beacon, a large rectangle could be detected. The rectangle is divided into two and the pixels in the top half are marked green and the pixels in the bottom half are marked pink. Even if some pixels were marked incorrectly it would not matter if enough samples were taken, as the tallying up process in the generation of training data would remove the skew data. The Newcastle University did some form of automatic colour classification in this year's competition.

Currently the robot is placed manually at certain positions in the field before a picture is taken. This requires that at least two people be present during calibration. As the pictures are always taken at certain spots in field, it is possible to write a strategy player file which would walk the dog automatically to certain spots on the field, face a particular heading and save a BFL image. However the dog's localisation obviously is greatly dependent on the vision and this method would only work if there is minimal change in the lighting, hence having minimal effect on the localisation routine.

A fundamental difference between the current colour detection algorithm and the way human's perceive colour is the use of a lookup table. Humans don't look at an individual spot to determine its colour, we look for patches with similar colours and detect edges and contrast to note change in colour. By using a colour lookup table the vision system will never be correct because it is possible for certain Y U V coordinate to be classified one colour in one picture and another colour in another picture. If a successful (and efficient) contrast or edge detection algorithm could be implemented there would be vast improvements in the vision system.

Chapter 3

Object Recognition

Section author: Benjamin Leung

3.1 Overview

Due the increase in field size, the effect of noise in object detection has exponentially increased. Beacons are not seen as often and localization information is scarce. By increasing the sensitivity of object detection, the occurrence of misidentification has also increased. The 2001 *sanity checks* are no longer sufficient in eliminating noise and its lack of organized structure and documentation made it difficult to understand and to add or remove sanity checks.

This section attempts to document the structure of sanity checks as of 2002.

In the rUNSWift 2002 code, object detection occurs in three phases:

1. **Physical Detection -** Detects whether the blobs physically resemble the object
2. **Logical Detection -** Detects that the object detected is logically placed in the view.
3. **Robot Transform -** using the distance from the camera and elevation of the object in the image - transform these coordinates into world model coordinates.

3.1.1 Additional 3D Transformation - Rotation of the image

The tilt and pan angle of the head causes the image seen by the camera to be rotated up to +/- 90 degrees. Since visual object detection revolves heavily around the vertical/horizontal alignment of different patches and the vertical elevation of the patches, the patches must be *un-rotated* so that their coordinates is relative to field.

The effective roll of the image from the combination of head tilt and pan is given by:

$$effective\_roll = asin(\sin(head\_pan) \times \sin(head\_tilt)) \tag{3.1}$$

By rotating the patches in the opposite direction (i.e. `-effective_roll` degrees) about the centre. The coordinates of the patches will become relative to the playing level of the field.

3.2 Visual Objects

After detecting each object, information relative to the camera is transformed into world coordinates or relative to the base of the robot necks.

The each object is described as the following:

- confidence factor - value range from 0 to 1000 (0 =¿ not seen), representing how reliable the statistic for this object is
- variance - variance of object determined from equation.
- centroid of the object (x,y)
- height, width, and top left corner (x,y) of the object bounding box
- height, width, and top left corner of the projected bounding box onto image due to occlusion, or limited view
- accumulated area of blobs of this object
- heading and elevation of the object in the image (pixel offset from the Y Axis and pixel offset from the X axis)
- elevation of the object relative to the base of the neck parallel to the ground
- distance from the object to the camera (used for ball tracking)
- distance from camera distance to the centroid of the object (used in calibration)
- distance from the neck to edge of the visual object
- two dimensional distance from to neck to visual object edge (parallel to the plane of the field)
- position of the object in three dimensional space relative to the neck,
- absolute direction the visual object is facing in world model
- orientation of the object (for robot)

3.3 Detection of Objects

3.3.1 Beacons

Beacons are the first objects to be identified and hence used as the basis for validating objects detected after it.

Beacon detection occurs in the following steps:

1. Locate pink blobs in the image;
2. Match pink blobs with nearby green/blue/yellow blobs;
3. Extract information about the distance and heading of the beacon based on dimensional information of the detected beacon; then
4. Use that information to validate the beacon.

Physical Detection

Beacons are detected in `LocateLandMarks()`. For each pink blob, nearby yellow/green/blue beacon is matched with it to check if it resembles a beacon. `CheckMatchingBeacons()` matches pink blob with a colour blob using the following constraints:

1. if either blob has zero area then ignore it
2. if the distance between the centre of the two blobs is greater than the sum of the half-perimeters of the pink and referenced blob then ignore it.
3. the ratio of the smaller blob to the large blob must be greater then 0.1
4. if the angle of the line to drawn between the two centers of the blobs is less than 60 degrees and greater than 120 degrees relative to the horizon then the two blobs are not sufficiently vertically align to be a beacon
5. if the lower blob of the beacon is surrounded by lots of green, discard the blob because it is most likely to be noise.

If a (green/blue/yellow) blob satisfies all the above constraint in respect to a pink blob, then it is passed off as a physical beacon. The type of beacon is identified by the elevation of the pink blob when compared to the blob that makes up the other half of the beacon. The height and width of the beacon is then scaled according to the dimensions of the largest blob out of the two. This is used to fix lighting problems or obstruction.

The distance from the camera to the beacon is based on the linear equation:

$$beacon\ distance = \frac{beaconDistanceConstant(= 1800)}{distance\ between\ the\ two\ centers\ of\ the\ blob} \tag{3.2}$$

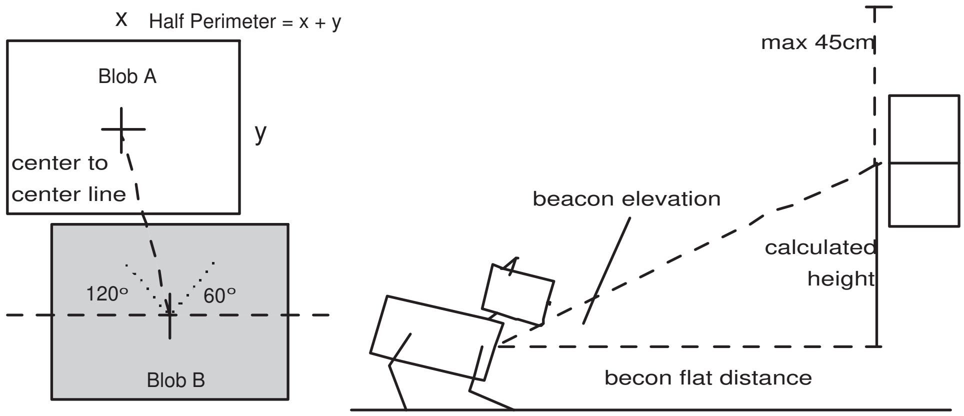


Figure 3.1: The various measures and method used in beacon sanity checks

The distance between the two centre also used to estimate the expected size of the beacons.

$$expected\ area = 2 \times (centre\ to\ centre)^2 \tag{3.3}$$

The ratio of the observed area with the expected area determines the confidence level of the detected beacon.

After calculating the distance, image elevation/heading and other information based on the dimensions of the beacon, these information are transformed into world model coordinates. The distance and elevation are modified to be relative from the base of the neck. Given that we have the have the flat distance to the beacon and the elevation of the centre, it is possible to work out the height of the beacon. This is effective in eliminating noise as the combination of the distance and elevation calculated from a spurious beacon usually don't correlate. The height of the center of the beacon is calculated as follows:

$$height\ to\ center = height\ of\ base\ of\ the\ neck + tan(beacon\ elevation) \times distance\ to\ beacon \tag{3.4}$$

The cut off used in the Fukuoka code was 45cm. This value was determined an experiment observing values detected by a dog walking towards and away from corner beacons diagonally across the field.

Special beacon sanity checks applied to the goalie, since localization measurements is extremely important to its performance. The two beacons that the goalie most often used to localize are situated on the opposite side of the field. At such large distance, the detected beacons are extremely small, and as such the distance calculated based on size is extremely sensitive to even 1 or 2 pixel changes. This problem is particularly evident when the beacons were seen closed to the edge of the camera. The warping in size and colour leads to miscalculations and fluctuating values that completely throw off the localization of the goalie. To solve this problem, beacons seen within a 10 pixel border of the image and with a calculated distance greater than 300 cm are discarded.

3.3.2 Goals

Ideally at the end of the beacon detection process, all blobs that belong to a beacon will be marked as *matched*. The remaining blue and yellow blobs must either be noise or portions of a goal. If there is a beacon observed in the image, all the blobs with centroid at an elevation higher (smaller y value) than the image elevation of that beacon is removed because no goal can be taller than a beacon.

After that, the assumption is that all blue or yellow blobs remaining in the image are portions of an observed goal if they satisfy a few sanity checks. The process begins by taking the first unmatched blue or yellow blobs and systematically adding other unmatched blue or yellow blobs until the either blob list is traversed. The following pseudo code describes the process:

```
for each unmatched goal colour blobs
{
    if (its area is zero)
        next blob;

    if (the image elevation of the centroid is greater than that
        of an observed beacon)
        next blob;

    if (the height of the blob is less than kGoalMinHeight)
        next blob;

    if (the width of the blob is less than kGoalMinWidth)
        next blob;

    if (the blob is left of the left goal beacon with confidence > 500 or
        right of the right goal beacon with confidence > 500)
        next blob;

    if (there are more than 10 unclassified pixels on the line
        drawn between the center of the current goal and the centroid of the
        blob)
        next blob;

    if (this blob is inside current goal bounding box)
        add blob to the goal object;

    if (there is no blobs in the current goal object)
        add blob to the goal object;
}
```

At the end of the for loop, extracted is a bounding box containing all the blobs that is considered a portion of the goal. If the height and width of this bounding box is bigger than a minimum constant and there is sufficient green pixels around it, then this bounding box is considered the goal.

Since the Aibo spends most of it time tracking or searching for the ball, it usually only sees the bottom half of the goal or the left/right portion of the goal. If the detected width and height of the bounding box is in a strange ratio, the height will be expanded vertically portioned to the observed width of the goal. In the 2001 code, expansion for the width also occurs but that was removed because it often expanded the wrong way (left as oppose to the right). Vertical expansion was not removed because it is necessary for distance calculation.

The distance to the goal is calculated based on its height and is determined by the equation (from 2001):

$$distance = 5192.11/(bounding\ box\ height - 4.1947); \tag{3.5}$$

After `RobotRelative()` is called, the logical sanity checks will be applied to the goal objects detected. Goal sanities are generally very obvious cases, e.g.

- goals that are too far away (greater than the diagonal distance of the field) are given zero confidence and infinite variance.
- if both blue and yellow goal is seen at the same time, assign zero confidence and infinite variance to the less confident one.

3.3.3 Ball

For information about ball detection and related sanity checks, refer to [Ola02]

3.3.4 Robot

Robot recognition is the most challenging and difficult problem of object recognition. Robot recognition can be broken down into two main problems:

1. grouping blobs into individual dogs, i.e. multiple dog detection within a single frame;
2. calculating distance and heading information to the detected dogs.

Multiple Dog Detection

Multiple robot recognition did not exist in the 2001 code. 2001 code detects the largest red or blue blob in the image and a bounding box is placed around that. It works well in most cases but often the largest blob does not represent the Aibo. The center of the bounding box does not necessarily represent the center of the Aibo. While this is currently not important, if passing or more advance game play strategies need to be develop, getting the heading of all the opponents observed on the screen will become more important.

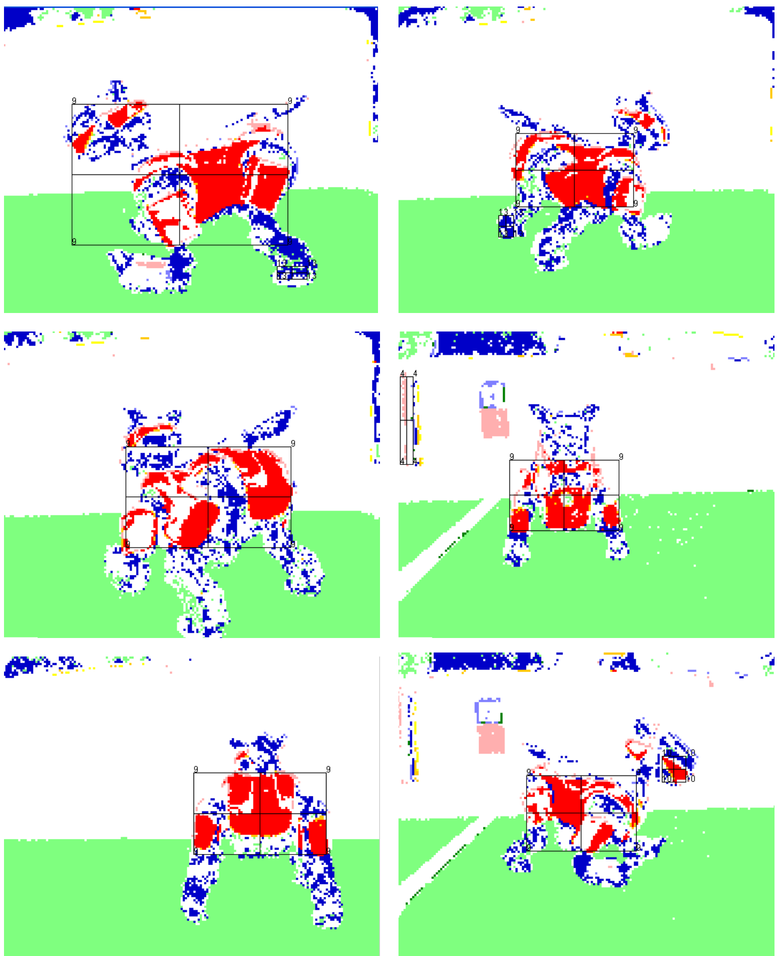


Figure 3.2: The non uniform arrangement of the patches and the changing bounding box size makes it difficult to separate multiple dogs and calculate the distance to them.

The heuristic used to group together blobs to form individual dogs is that the horizontal extremities of the patches that belong to a dog always overlap or are at most separated by a small distance. Blobs that a too far apart horizontally most likely belong to different dog.

In order to utilize this heuristic in every orientation of the head, an extra transform must be performed on the image. Equation 3.1 is applied to all the red and blue blobs to ensure their orientation is in line with the field plane. The usual noise filtering process applies, all blue blobs and red blobs with elevations greater than the top of the beacons and goals are removed.

The following pseudo code separates the blobs into a maximum of four dogs:

```
\\These four values are the extremities of the robot bounding box
xMin = 0; xMax = 0; yMin = 0; yMax = 0; buffer = 0;

for each blue/red blob (order of centroids from left to right) {
    if (blob area > 0) {

        if ((the left most edge of the blob <= xMin Or
             the left most edge of the blob <= xMax + buffer))
        {
            then expand bounding box to include this blob;

        }
        else
        {
            // this case should only be entered when
            // there is a break in the x distance between
            // the dogs.

            assign the current bounding box dimensions to current
            dog visual object;

            all the blobs now belong to the next dog;

            set xMin, xMax, yMin, yMax to the edges of this
            blob, forming a bounding box around this outlying blob;

        }

        // the width of the bounding box containing a dog standing
        // perpendicular to the camera
        double hor = 26.0;

        // the height of the bounding box containing a dog
        standing
        // perpendicular to the camera
        double vert = 18.0;
```

```
        // buffer is the maximum distance to look right

        // max_width is the maximum length of a bounding box
        // (ie bounding box of the dog viewed from the side)
        // given a specific height. In this case, max_width
        // is the maximum possible length of a bounding
        // box with the height of the current bounding box.
        max_width = (current bounding box height/vert)*hor;

        buffer = MAX(MIN(max_width - current_width, 20), 10);
    }
}
```

At the end this for loop, up to four bounding boxes representing the location of possible dogs can be extracted.

Given their better visibility when compared to blue dogs, red dogs are detected first. At the end of the red dog detection process, all blue blobs with centroids within the bounding boxes of the red dogs are removed as sometimes the black regions of the robots are classified as blue under certain lighting conditions.

After that, the same process is applied to the blue blobs for blue dog detection .

Robot Distance Calculation

Given the dimensions of the bounding box, it is now possible to calculate the distance of the robot to the camera. Due to the non uniform shape of the robot when viewed from different angle, it is difficult to apply the standard bounding box method to resolve the distance. The method used in the 2002 code uses a relationship between the angle of deviation of the object and the flat distance of the object from the camera . Robot distance is calculated via the angle of deviation of the bottom edge of the bounding box from the field plane. The benefit of using this method is that the distance to a robot can be calculated without seeing the whole object. Since the Aibo is expected to spend most time tracking the ball, it generally only sees the legs patches of other robots. Using this triangle method, distance calculation based on seeing only the bottom 50% of a robot is identical to seeing 100% of a robot.

While Equation 3.6 is theoretically correct, the actual relationship between deviation and distance do not precisely lie on this equation. In order to provide a more accurate distance measurement to the world model, the tangent mapping between distance and angle of deviation was replaced with equations (Equation 3.7 and Equation 3.8) derived from using regression on a sample of observed distances and angle of deviations. This proved to be more accurate in determining the distance to the robots.

There are separate equations for red and blue robots because their visibility is quite different. Red blobs are much more defined than the blue blobs. Reliable

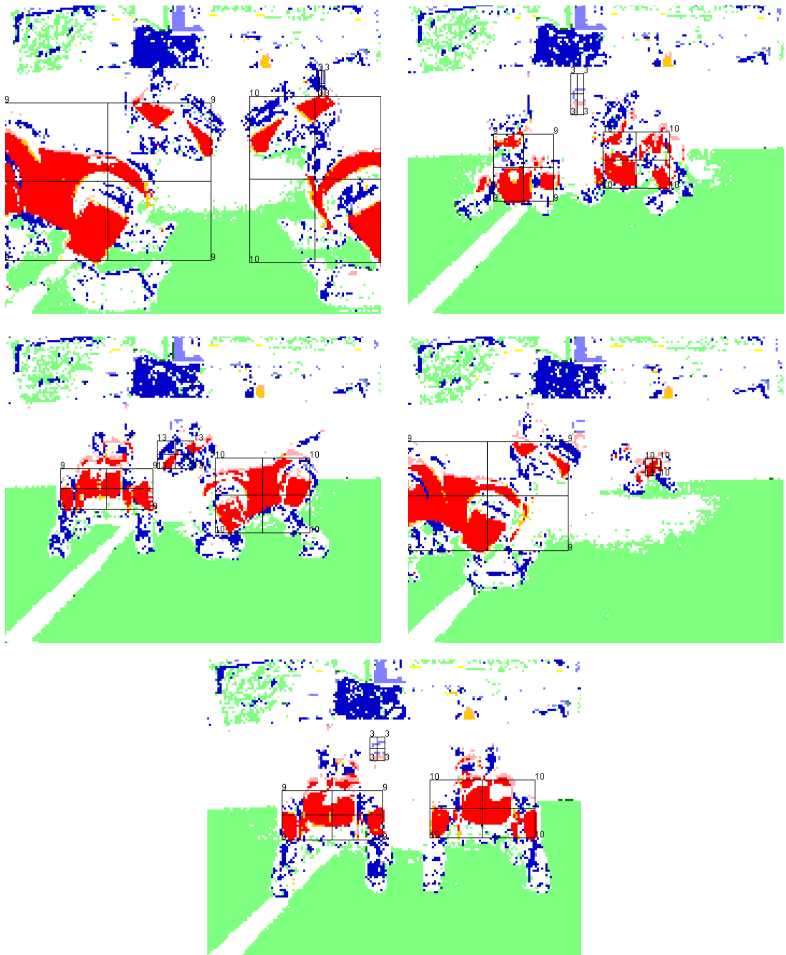


Figure 3.3: By applying the algorithm, blobs can be separated into multiple dogs.

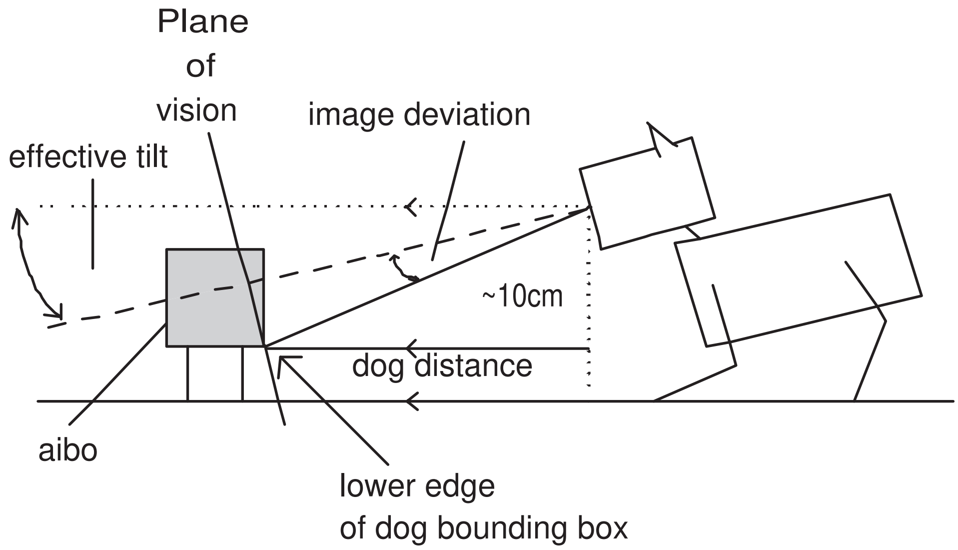


Figure 3.4: Theortical relationship tangent relationship between the angle of deviation, camera height and robot distance (Equation 3.6).

$$distance \approx \tan(deviation\ angle) \times 10 \tag{3.6}$$

distance measurements based on vision for a red dog is up to 2.2m whereas it is only 1.2m for a blue dog.

After `RobotRelative()` is called to transform all the values, standard sanity checks such as discarding dogs above goals and beacons and eliminating robot at great distances are applied to ensure robot objects are in logical positions.

Calculating robot variance

Variance and confidence level for robot objects is calculated differently to other visual objects since there is no expected area or other constant values. Confidence level is based on the height of the bounding box proportion to the screen (176 x 144). Height is probably the most invariant property of a robot as it is viewed from different angles. As the robot gets closer to the camera, it height increase and as such, confidence in its distance increases.

Variance for a robot is calculated using the average of the distance measured if the bounding box is 2 pixels lower (i.e. closer to the robot) and if the bounding box is 1 pixel higher (i.e. further from the robot). The difference in the number of pixels used when lowering and lifting the bounding box is to balance out the hyperbolic relationship between the angle of deviation and the distance.

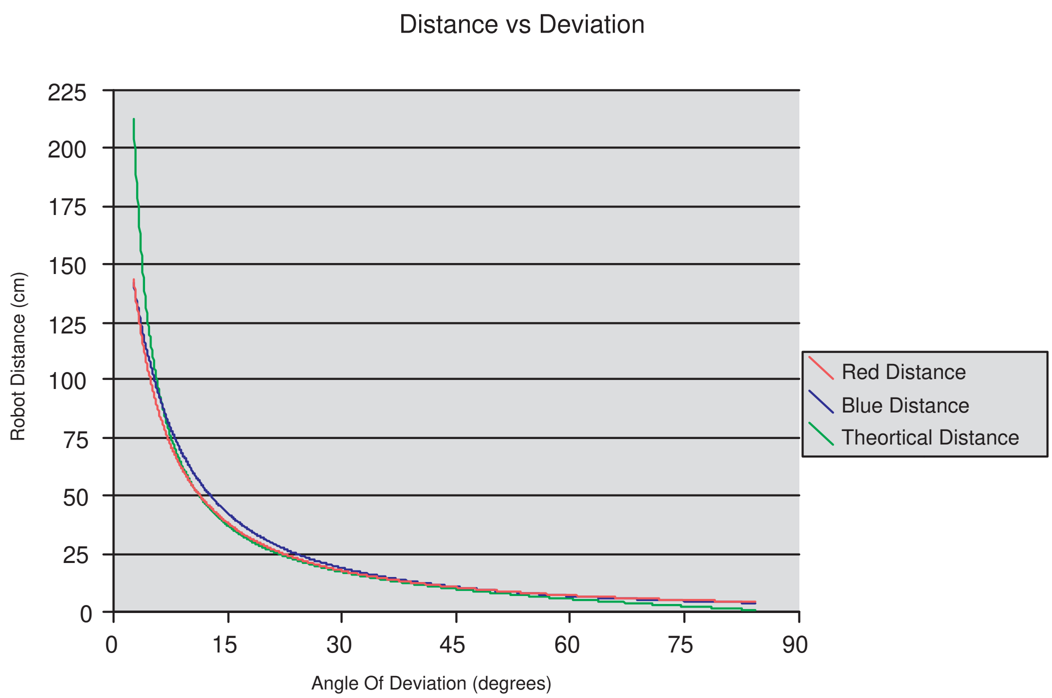


Figure 3.5: A graph comparing the different equations used to calculate dog distance. It can be seen that the equations generated via regression deviate (very slightly) from the theoretical tangent relationship

The equations used to calculate the distance of a red Aibo and a blue Aibo:

$$\log(red\ robot\ dist) = 0.890680 - 1.37435 \times \log(dev) - 0.316338 \times \log(dev)^2 \quad (3.7)$$

$$\log(blue\ robot\ dist) = 0.879319 - 1.54566 \times \log(dev) - 0.441522 \times \log(dev)^2 \quad (3.8)$$

3.4 Limitation and Further Work

There can be much improvements to object recognition. The current blob matching method of object detection works quite well in most cases, but it is still to sensitive to changes in lighting conditions. With more powerful CPUs in the Aibo's, perhaps learning algorithms or image processing algorithms can be applied to the raw camera feed to provide a more robust method of object detection. Learning could also be used on various heuristics about the positions and size of different patches to assist in object detection.

Robot distance measurement is very accurate for a stable camera. In case of a walking dog during game play, the shaking of the camera causes the distance values to fluctuate quite wildly. Perhaps in future, parameters from pwalk can be applied in calculating the exact position of the camera in 3D space, giving a more accurate deviation measurement.

Chapter 4

Localisation

Section author: Zheng David Wang

4.1 Introduction

One of the most difficult tasks in creating the software to control the robots is localisation. That is, by observing landmarks around the field, the robot must be able to estimate its own position within the field and then estimate the positions of the ball and other robots. The source of difficulty in localisation is the inaccuracy of the sensors. The colour CMOS camera is the robot's main sensor. Since the robot only has one camera, distance to a landmark is estimated by its apparent size. With an image resolution of only 176 x 144 pixels, distant objects may occupy only a few pixels, so a difference of one or two pixels can make a big difference to the distance estimate. Noise in the camera image almost guarantees that there will be pixels that appear the incorrect colour and therefore the size of the object and its distance is always uncertain. So the question is: how can the robot function reasonably reliably with uncertain sensors.

Information from the robot's vision system, as well as odometry from the locomotion module are combined to create a world model. Since the dimensions of the field are known in advance, the world model is constructed by placing the mobile objects on a map of the field. The world model stores the objects x, y location and heading on field. As objects are observed to move, their positions and heading in the map are updated. However, some apparent motions may only be noise in the image. It is therefore unwise to instantaneously update the position based on a single measurement.

In 2002, a new addition to the competition was the availability of wireless network communication between the robots. This allows robots to share information about the world. In theory, one would think that this is a good idea since, for example, a robot that is unable to see the ball may still know its location because a team mate that sees the ball can broadcast its world model. Suppose however that two robots can see the ball, but due to the inaccuracy of their sensors, each has a different estimate of the ball's location. Which should be believed? Can two uncertain measurements be combined sensibly? In this paper, we address these problems and describe our experiences in attempting to solve them for a domain that contains eight autonomous robot interacting and interfering with each other in often unpredictable ways.

This chapter will first introduce the old localisation and address its short comings that motivated new developments. This improvement to the world model is based on Kalman filter. This will be examined next. Following this, the vision module modifications to cater for the new technique will be described. Once these fundamentals of the localisation technique is briefed, we will address various localisation techniques including, 2 beacon localisation, single beacon localisation, goal localisation and active localisation. Further more, the localisation of other objects on the field will be described, especially the new development that allowed multiple opponents and team-mates to be identified on the field. Then, the techniques for using the wireless for localisation will be explained. Finally, an evaluation of the localisation will be conducted.

Terminology
Before we dive into the details of localisation, the terms *Vision Model*, *World Model* and *Wireless Model* needs to be explained. Due to the introduction of wireless the localisation module is becoming more complex. To conceptually cater for this complexity we talk about models of the world in 3 forms. The *Vision Model* is a instantaneous model of the world direct from the vision module. In other words, it's what the robot sees. The *World Model* is a model that is derived from the continuing combination of the vision model and local motion feed backs. More simply, it combines the new information that the robot sees with what it previously saw. The *Wireless Model* is a model of the world that combines the world model of all the robots. The world model of other robots are received via the wireless module.

4.2 Old Localisation and Motivations

4.2.1 Old World Model (2001)

The update method adopted was to compromise between the objects current position in the world model and the perceived position in vision model by gradually "nudging" the position away from the current position towards the perceived position. An *ad hoc* confidence factor was attached to the location of an object in the world model and the vision model. This described a belief in the accuracy of the estimated distance and heading to the object. The confidence factor was based on the percentage of the object seen and other *ad hoc* criteria. It was used to determine the weight of the contribution of the object's perceived distance to the update of the estimated position of the object.

The update algorithm is described in Figure 4.1. This algorithm is simply a

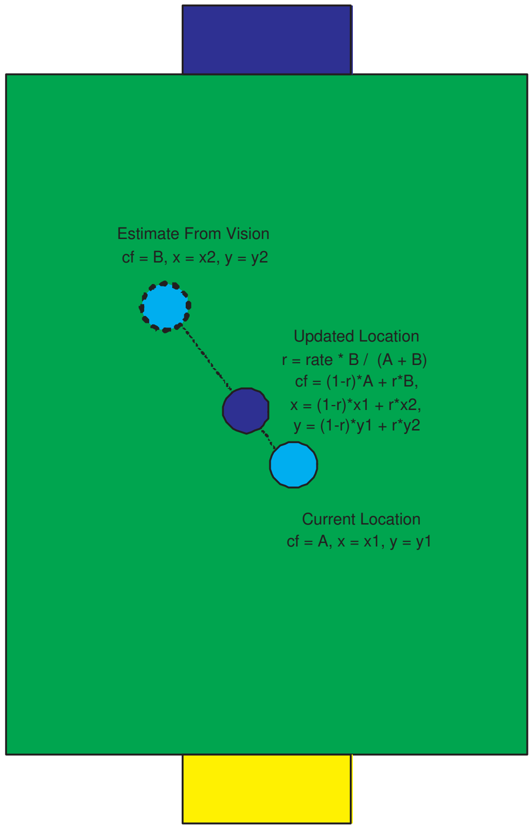


Figure 4.1: Old World Model Location Update

weighted average of the new estimate with the old estimate. A rate factor is used in the algorithm to slow down the attraction to the new estimate. This reduces jitters from noise in measurement.

The problems with the old localisation technique can be revealed by looking at how a robot localises itself. A robot localises itself by using the occasional glimpses of beacons around the field. Even though the vision system is unable to steadily determine the distance of a beacon more than 2 metres away. This localisation method worked well in previous competitions. Last year's team overcame these short comings by ignoring far beacons, because size of the field is small enough to allow close beacons to be seen frequently. Unfortunately, in 2002 the size of the field was increased from 2 x 3 metres to 3 x 4.4 metres. This meant the chance of seeing a good accurate close beacon is more difficult and continue to throw away localisation information such as far beacons is not helpful. For example, on the small field, if the robot is placed at the left side of the field close to the wall. Then it will still be able to observe a relatively good (accurate) beacon by looking left or right of the field. The distance to beacons will be less than 2 metres. On the other hand, with the large field, if the same situation happen the robot will only use the information from the left side of the field. We have to use every bit of information possible for localisation. Using noisy measurements from far beacons caused the *ad hoc* method based on confidence factors in previous years broke down. By simply removing the beacon distance constraint we tested the old localisation using the wireless debugging tool we developed. This tool allowed us to see the live world model of the robot on a computer. In this experiment we found the robot jumped unpredictably in the world model when it is at one end of the field and looked towards the other end.

$$rate = \begin{cases} 0.4, & curCF > 600 \\ 1, & otherwise \end{cases} \tag{4.1}$$

Figure 4.2: Rate Change Condition

The old localisation used many *ad hoc* values such as confidence factors, update rate and the condition to use a different update rate (figure 4.2). It also relied on the beacon measurements to be accurate and noise free. This is no longer the case as we want to use every bit of localisation information. One way to solve this is to make far objects have smaller confidence value. The can be done by using another *ad hoc* scaling on the confidence factor to make confidence value distance dependent. However, this would be another hack on top. There has to be a better way. By closer examination of the old localisation, the method actually looks very similar to Kalman filters. So why don't we use Kalman filters. This is the standard technique used in many existing systems such as missile guidance. Kalman filters uses variances which is statistically gathered and can solve the problems such as the *ad hoc* values and can make the estimates more steady with noisy data as statically collected data will reflect the reality and the distance dependence of the observed objects.

4.2.2 Old Vision System

The vision system identifies objects that it sees in a camera frame by using colour coding. It estimates the distance and heading to the object and attaches an confidence value to the observation. This confidence value describes the belief on the data. Under the old vision system, a confidence value only use the ratio between observed size of the beacon and the expected size of the beacon at that distance. The observed size is the number of pixels categorized as part of the beacon i.e. the mass. The expected size is the number of pixels that are expected to be categorized as the beacon, i.e. the bounding box that is scaled to have the same ratio as the actual beacon. see figure 4.3

$$cf = \frac{mass}{ExpectedSize} \times 1000 \tag{4.2}$$

Figure 4.3: Object Confidence Factor Determination

The flaw to this scheme is that far beacons (> 2 metres) only occupies a few pixels. It can be as little as 10 pixels. This means a small change in the number of pixels can dramatically change the confidence factor. At that distance a few pixels will also mean a big difference in distance. It is not unusual to miss classify beacon colours. This can make far beacon seem smaller/further than reality. In addition, especially with a noisy background, a few pixel of noise that is categorized as the same colour as the beacon is also not uncommon. These noise can make the beacon larger/closer. Since, far beacons is observed in the robots camera only as a few cluttered pixels. This means the bounding box that is proportional to the real beacon almost fits the beacon exactly. In result, observe equation in figure 4.3, the beacon distance will have very high confidence. Of cause we all know in reality it is the opposite. Seen all these problems, how can we solve it elegantly? Kalman filter will solve this because it will use statistically based measurements. Hence, the confidence will automatically depend on distance because when you compare the actual and estimated distance there will be more noise as a beacon gets further. This turns into higher variance in Kalman filters.

4.3 The Kalman Filter

In 1960, R.E. Kalman published his famous paper describing a recursive solution to the discrete-data linear filtering problem. Since that time, due in large part to advances in digital computing, the Kalman filter has been the subject of extensive research and application, particularly in the area of autonomous or assisted navigation.

The Kalman filter is a set of mathematical equations that provides an efficient computational (recursive) solution of the least-squares method. The filter is very powerful in several aspects: it supports estimations of past, present, and even future states, and it can do so even when the precise nature of the modelled system is unknown.

For the our purpose we have used the discrete Kalman filter. Hence, in our explanation our reference to Kalman filter means discrete Kalman filter. In Kalman filter, there are two sets of equations: time update equations and measurement update equations ([WB01]). The time update equations are responsible for projecting forward (in time) an estimated state, and are used when odometry information is received from the locomotion module. The measurement update equations on the other hand are responsible for feedback by incorporating new measurements and are used when information from the vision system is received. The two set of equations form an ongoing discrete Kalman filter cycle (figure 4.4).

For the purposes of exposition, we will briefly explain how the Kalman filter works by describing how a robot can localise itself relative to fixed landmarks. This will explain the time and measurement update process. For full details of Kalman filter I refer you to [WB01]. The variables for robot localisation are the x and y coordinates, the variance of the location, the heading and the variance of the heading. The origin is at the bottom left corner of the field. The x axis runs along the edge of the field with the robot's own goal below the axis and the opponents goal above the axis. The y-axis runs along the left edge of the field with the goals to the right of the axis.

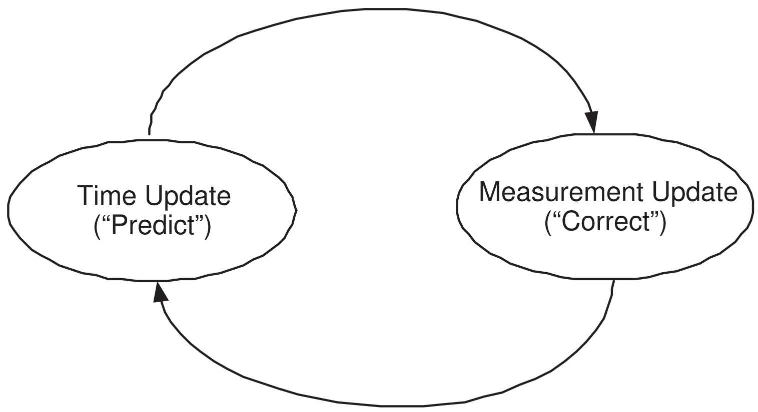


Figure 4.4: Kalman Filter Cycle

Robot localisation uses beacons and goals as the landmarks and receives odometry from the locomotion module. When the robot sees a beacon or a goal, the identified object is passed to the world model. This module then uses the position measurements and the current estimated position and heading to build a new position and heading where it believes the robot should be. The robot first checks if it can see two beacons. If it can, it uses the two beacons to triangulate its location and update using Kalman filters. If it can only see one beacon, it will use the Kalman Filter to move its estimated position towards the beacon so that the world model distance to the beacon is closer to the measured distance.

4.3.1 Measurement Update

The measurement update equations are responsible for feedback by incorporating new measurements into the estimate. In the robot, once the distance and heading measurements are obtained, they must be combined with the current estimates of location and heading to derive the new state. At this point, the Kalman filter's time measurement equations are applied (figure 4.5), assuming we are trying to estimate x. $\hat{x}$ is the estimate of x.

$$K_k = \frac{P_k^- H^T}{H P_k^- H^T + R} \tag{4.3}$$

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-) \tag{4.4}$$

$$P_k = (I - K_k H)P_k^- \tag{4.5}$$

Figure 4.5: Discrete Kalman Filter Measurement Update Equations

In the discrete Kalman filter measurement update equations, values with a "super minus" are those obtained after the time update phase. $\hat{x}_k$ is the estimated (vector) value at time k. P_k is the variance at time k. H is the matrix that relates the measurement z_k to the prediction $\hat{x}_k$. In other words it's the matrix that changes the units from measurement to the estimate. R is the measurement noise (variance). K_k is called the Kalman gain. Notice these equations uses matrixes. This is because, Kalman filter can be used for values (x)of any dimension.

For our task, we have decided to use a one dimensional Kalman filter for object localisation by assuming the variance of the objects in both the x and y direct are the same. We have also assumed the heading is independent hence, it is another one dimensional Kalman filter. Choosing a higher dimensional Kalman Filter is possible however, during a game a robot can easily have its leg caught on the side walls or it can be pushed around by opponents. These unpredictable and dynamic environment means a more complex solution will not always be better as they will place more assumptions. In addition, we have taken the approach to start small. This will allow us to debug and test in the tight time frame of the competition.

$$K = \frac{var}{var + R} \tag{4.6}$$

$$x = x + K \times \triangle x \tag{4.7}$$

$$var \leftarrow (1 - K) \times var \tag{4.8}$$

Figure 4.6: Simplified Measurement Update Equations

Figure 4.6 is the simplified discrete Kalman filter measurement update equations for our task. This equation shows how an object's x value is updated. In figure 4.6 var is the objects variance; R is an estimate of the measurement noise (usually determined experimentally prior to the measurements); K is called the Kalman gain. In our simplified task, H is removed because $H = 1$ as the measurements will already be in the units of the estimate. First, the gain is updated. Then, we take the difference, $\triangle x$, of the current estimate and the measured value and use it to adjust the estimate. This is the component $(z_k - H\hat{x}_k^-)$ in figure 4.6. The update equations above are repeated for the robot's y position and its heading. For updating the heading we have taken care to ensure the circular behaviour of angles in applying Kalman Filters, i.e. 0 is the same as 360. The measurement noise, R, is calculated using statistical information on estimated distance of the object. How R is obtained in is discussed further in Section 4.4.

4.3.2 Time Update

This projects the current state estimate ahead in time. This is done when the robot obtains the odometry measurements from the locomotion module to help it localise. This is useful because it allows the robot to know roughly where it is even when it has not seen a landmark for some time. However, its confidence in its location should decreases very rapidly. To model this process, the Kalman filter's time update equations are useful (figure 4.7). These equations are executed in sequence.

$$\hat{x}_k^- = A\hat{x}_k + Bu_k \tag{4.9}$$

$$P_k^- = AP_{k-1}A^T + Q \tag{4.10}$$

Figure 4.7: Discrete Kalman Filter Time Update Equations

In the time update equations from figure 4.7, A is the motion matrix that predicts the objects location from its motion. B is the matrix that relates the optional control input u_k to the prediction. It converts the u_k to the units used by x_k. Q is the process noise covariance. This is the noise of the prediction.

$$x = x + \triangle x \tag{4.11}$$

$$var = var + Q \tag{4.12}$$

Figure 4.8: Simplified Time Update Equations

Again, for our simplified one dimensional Kalman filter, we have only used the equations in figure 4.8. This is derived from equations in figure 4.5 by setting $A = 1$, $B = 1$ and u_x to $\triangle x$. $A = 1$ because the assumption we can't predict where the robot will be. This way we simplify the problem. $B = 1$ because the movement estimated by local motion is already in the same units. The variance is updated by the process noise, Q. Q is estimated before hand. In the case of the legged robots, this is an experimentally determined estimate of the amount of slippage when the legs attempt to move the robot. This Q value is actually increased to take into account the fact a robot can be in a scrum or it can be picked up by a referee and moved around. In those cases you want the robot to move its estimate to the new location as quickly as possible.

4.4 Vision System Modification

To cater for Kalman filters the vision system must produce a variance measurement for each object identified. In order to escape from ad hoc values like the confidence factors, the variances of measurements are derived by recording the error in the estimated distance. By using statically collected variance measurements not only have we moved away from magic numbers, we have also made the confidence of distant measurement distant dependent. This means far objects will have much higher variance than those of closer objects due to the physical limitations of the camera. This should make the world model much more stable and resilient to error from far beacons.

To illustrate how the variance of distant measurements is obtained, we will demonstrate the process of constructing a variance equation for beacons. First, we place a stationary robot on the field looking at a beacon and record distance estimates at various distances. They are recorded in increments of 5 cm when close to the beacon and 10 com increments when far. We have taken 450 samples for beacons. Next, with the collected data, we construct a graph (Figure 4.9),

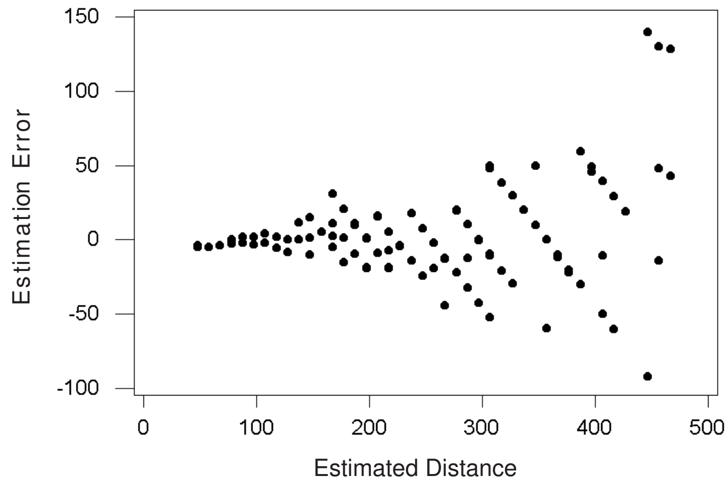


Figure 4.9: Error In Beacon Distance

which shows the behaviour of the error. By understanding the rough behaviour of the error, we can determine how to model the variance. From this graph, we can see the error increases over distance and in a exponential fashion. Hence, we need a exponential function to model the error. By finding a line of best fit for the error, we can obtain the variance equation by simply squaring this line. Subsequently, to find the line of best fit, we took logs of the error and then apply regression over the new values. Figure 4.10 shows the results of the regression using a statistic tool —minitab. Finally, the equation for the variance of beacons is obtained by taking the equation from regression, invert the log function then square the result. The resultant equation for variance is $var_{beacon} = (10^{0.0203926+0.007738 \times EstimatedDistance})^2$

With the above additions, the vision module is now able to produce a variance for each object recognised. In practice, because objects can be partially occluded by other robots or they can be partially seen at the edge of the camera, an extra variance that is inversely proportional to the percentage of the beacon seen is added. The percentage of beacon seen is the mass of the beacon in camera frame divided by the expected size (see figure 4.3)Though this seems to be fudging again, what is important is that a relationship between the distance of the observed object and its confidence is established.

The time update process actually is carried out right before the measurement update procedures in the code rather than doing after every leg motion. This is because there is a one frame difference between the number of times the local motion update and camera frame is received. This procedure ensure the

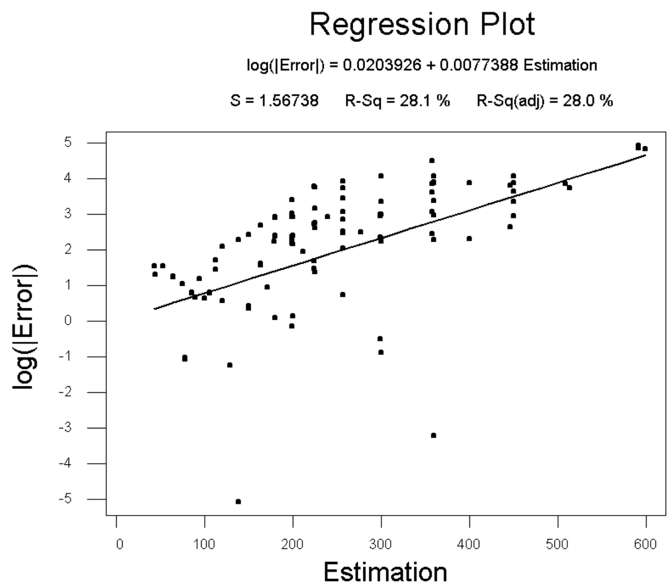


Figure 4.10: Regression on Error In Beacon Distance

processes are in sync.

4.5 Beacon Localisation

4.5.1 2 Beacon Localisation

The 2 beacon localisation routine has remained largely untouched since UNSW1999 team. This routine is invoked when the robot sees at least two beacons with high confidence. Small adjustments were made this year to the update procedures. This is modified to use the Kalman filters.

In the UNSW2001 code, the triangulation used was based on perceived distance to 2 beacons, the identity of the 2 beacons, and their perceived angular displacement. The algorithm will use the distance of the closer beacon and triangulate if the other beacon is too far away. Otherwise it will use similar triangles to adjust the distance to the 2 beacons until the perceived distance between the two beacons match the true distance between the beacons on the field. The output of the 2 beacon localisation routine is the new estimated instantaneous location and heading of the robot on the field based on the observed beacons and the confidence factor of this estimate.

This year variances are used instead of confidence factor. Consequently, the output of the procedure needs to be modified to produce a variance. Because this output will be later be used as the measurement value (z_k, figure 4.6) in the correction process in Kalman filters, we should understand the variance produced is the uncertainty of measurement. Since the estimate is based on beacons, the variance is justified by the uncertainty of the beacons seen. Last year, the output is a confidence factor calculated using the harmonic mean of the 2 beacons' confidence factor. This year, 2 options were considered to produce the variance output: continue to use harmonic mean or use the measurement update equations as those in Kalman filters by thinking one beacon is updated before the other (see equation 4.6, 4.7 on page 57). The harmonic mean equation is show in figure 4.11. By solving the equations for Kalman filters measurement update the derived equation is almost exactly the same, except it did not have the multiplying factor of 2 in the numerator.

$$var = \frac{2 \times beacon1.var \times beacon2.var}{beacon1.var + beacon2.var} \tag{4.13}$$

Figure 4.11: Measurement Update Equations

After a number of tests by placing the robot on various parts of the field and seeing the world model display, no obvious advantage can be seen between the 2 techniques. If we used Kalman filter's equation the resultant variance will be smaller when the 2 beacons have the same vairance. This makes conceptual sense since we have more beacons to tell where we are we should be more confident. However, considering the fact on the field size is increased reducing the chance of seeing 2 good close beacons and there was no significant advantage, the harmonic means is used because it produces a larger variance and hence was a safer choose in avoiding jittering localisation.

4.5.2 Single Beacon Localisation

Single beacon localisation was incorporated into the localisation module in year 2000. It readjusts the robot's location gradually based on one observed beacon and the robot's estimate of its current location. This routine is quite useful because while the robot is contesting the ball in match-play, often only part of a beacon can be seen. The robot would lost a lot of precious time to look around to localise if 2 beacons were always required. This routine is based on the assumption that the estimate of robot's location is fairly accurate already and the occasional observation of single beacons can be used to confirm and slightly readjust the estimate.

The belief on the estimate in the previous localisation technique is simply the confidence factor of the beacon used for this localisation routine. This year, the output is swapped for the variance of the beacon used for the localisation which will latter used in the Kalman filter to update the estimate.

4.6 Goal Localisation

This routine is applied when the goal is the only object that can be seen. In this case the robot adjusts its heading based on the perceived heading of the goal. Because the perceived distance to the goal is generally inaccurate for a number of reasons, adjustment to the location of the robot is discarded in the technique.

This is a topic that was investigated by all the previous teams and none was able to successfully establish a solution to localise using the goal. This year we have improved on the techniques used by the past team and explored many other techniques. However, given the time constraint a successful solution is not yet obtained, but I would like to describe the attempts we have made and the possible solution for future development.

4.6.1 Distance Estimation Improvement

The distance to the goal is predicted using the height of the observed goal. The height is used because the width of the goal can be easily obstructed by the opponent goalie. The taller the observed goal the closer we are to it. Unfortunately, the distance measurements from previous years were always quite poor. Using the data capturing tool we developed to capture the error in distance measurement for the beacon, we captured the height of the bounding box of the goal at distances intervals of 10 cm and feed the data through a statistic tool (minitab) and found an equation that was able to estimate the distance to the goal more accurately. We obtained the variance at the same time. The distance estimation to the goal is quite accurate when the robot is approximately with in 2 metres from the goal. This can be further improved if more samples were taken and in at more dense intervals.

The improved distance measurement gave us optimism. However, it was not used in the competition because, the chances of a robot in seeing the full height of the goal is quite slim. When the robot is chasing after a ball, it will look down and focus on the ball. In this situation the robot will only see the bottom part of the goal. Even though there are measures taken to use the width of the goal seen to estimate the height, the result is not brilliant in game play because, the probability to see the unobstructed full width of the goal is low. If the robot uses this information then it we be wrongly placed. In another situation, when the robot is right in front of the opponent goal, where most likely it will be competing for the ball with the goalie, the chance of seeing the full goal is more slim. Under numerous game play testing we found simply improving the distance measurement did not solve the problem.

4.6.2 Goal-Beacon Localisation

The blob of the goal seen in a camera frame provides a lot of information such as the width, height, shape, slant of the edges. Until now there isn't a way to successfully utilise the information by itself. Goal distance is not reliable in game play; beacon distance measurement is poor once the distance when it's more than 2 metres. Perhaps we can combine the 2 piece of information.

In this method the goal and a beacon is used to triangulate the location of the robot. The information used is the observed distance to the beacon, the identity of the beacon and the observed inclusive angle between the centre of the goal and the beacon. Since we know the real distance between the goal and the observed beacon, using these pieces of information we can calculate the location of the robot on the field using the equations in figure 4.6.2 and diagram in figure 4.13.

$$kown : d1, d3, d4, bx, by, \theta$$
$$find : x, y$$

$$d2 = \sqrt{d3^2 + (\frac{1}{2}d4)^2}$$
$$\beta = \sin^{-1}(\frac{\frac{1}{2}d4}{d3})$$
$$\alpha = 180 - \sin^{-1}(\frac{\sin\theta \times d1}{d2}) - \theta - \beta$$
$$x = bx + \cos(-180 - \alpha)$$
$$y = by + \sin(-180 - \alpha)$$

Figure 4.12: Beacon-Goal Localisation Equation

This technique worked well when the robot is in the approximate location as in figure 4.13. There are a number of difficulties with this technique. First, due to the size of the field, it is difficult to get a full font view of the goal that will have the full goal width and the beacon. The goal can be on the edge of the camera or obstructed by an object. This will affect the angle between the two object. This method also depended heavily on the distance to the beacon. The distance to the beacon at that range is approximately 2 metres. This is the where the distance estimation of the beacons starts to deteriorate. Also to have both the goal and beacon in view, the beacon will be seen at the edge of the camera where it will distorted (elongated) by the camera lenses. These conditions made this technique difficult in game, hence it was not used.

¿From this technique we did learn something which can be used in the future that can solve a weakness in our localisation. Presently, the localisation technique is unable to accurately determine the y value of the robot when the robot is in the penalty area near the wall facing the near by goal. By using only the beacon information localisation constantly underestimates the distance to wall due to the noise and errors in beacon distance estimation. Being able to know that you are near the wall of the opponent goal is especially important for the robots to choose the right edge behaviour and it is equally important for the

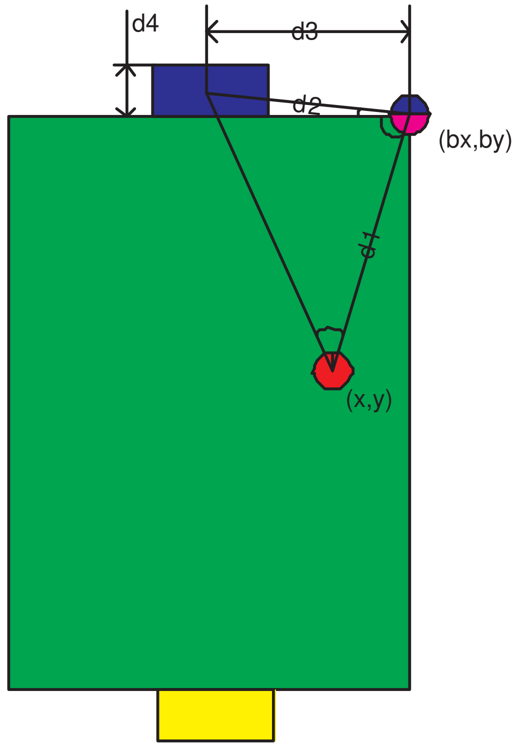


Figure 4.13: Beacon-Goal Localisation

goalie to know that information so they can attack the ball along the edge of the field. In these situations the beacon and goal localisation is actually very useful in describing the y value of the robot. The goal-beacon localisation is slightly different in this case because the robot is at the side of the goal, since the outer wall of the goal is not colored, the robot will not be able to a full frontal view of the goal to get the full width. Hence, the edge of the goal that is closer to the observed beacon is used instead. The x value of the procedure should not be used for the reason that it will be extremely sensitive to tiny angle differences between the objects. Under some initial tests this method was quite successful. This was discovered very close to the competition, therefore we didn't have enough time to fully implement it. This is definitely worth thinking about in the future.

4.6.3 Active Localisation

This year, the size of the field has doubled. This reduces the chance of seeing a close beacon. Under these circumstances, the team has taken the philosophy that catching glimpse of the beacon while chasing around the ball is not enough. To help the robot localise better on the larger field active localisation is adopted. In this method the robot will occasionally look for beacons while chasing the ball and it will hold the ball and localise in the opponents penalty area.

Quick look is a technique used when the robot is chasing the ball. It will regularly look up towards the direction where it expects the beacons. This will happen if the robot is looking at the ball, its more than 30 cm away from the ball and it has not seen a beacon for the past 25 camera frames. These conditions ensure the robot will be relatively well localised before it gets close to the ball. It also ensures the robot will not loose track of the ball when looking for beacons by using the 30 cm constraint.

As discussed in the goal and beacon localisation (see section 4.4), localisation is especially poor when the robot is inside the penalty area near the goal and facing towards the wall where the goal resides. This is the region where fine action is required. A kick in the wrong direction can cause our robots loose control of the ball. The presence of the opponents goalie will also make recovery difficult. Hence, we took on the approach that its better to be right than to be fast and recover the ball later. This issue is solved by making the robot grab the ball and then pan its head when it is in that region and it is uncertain of its location. The robot will first pan towards the direction where it expects the goal. It will stop the procedure when its variance is reduced, or it saw the goal or it has panned in both directions. By taking these approaches, wrong kicks or turns has been avoided in the penalty area.

4.7 Multiple Team-mates and Opponents in the World Model

Prior to 2001, the world model didn't contain any robot apart from yourself. The UNSW 2001 team added 2 robots in the world model that represented a team-mate and an opponent. In that year there are only 3 players on each team. This model was sufficient given the chance of seen 2 robots of the same team will be low. They were also constrained by their vision system which was unable to reliably detect more than 1 robot. In this years competition, there are 4 robots in each team. This increases the chance of seen 2 robots. Having more than 2 robots in the world model also allows better strategy planning. There were also a major improvement in the vision module this year that successfully recognises more than 1 robot from the same team in a camera frame. In addition, by using the elevation of the bottom edge of the bounding box of a robot more accurate distance measurement were obtained. (see Robot Recognition)

Given these changes in game condition and the improvement in vision, the world model is upgraded to be multiple team-mate and opponent capable. The world model stores information on 4 team-mate and 4 opponent robot. The team-mate information stored does not include the robot itself. The extra team-mate block is there to reserved space for future development. Kalman filters is used to update the robot. This follows the measurement and time update phase. The subsections below will explain the processes.

4.7.1 Measurement Update

As well as localising itself, a robot must also estimate the positions of the ball and the other robots. The Kalman filter update algorithm is the same. Updating the location of the ball is easy, there is a 1-to-1 correspondence between the world model ball and the vision model ball. However, a complication exists in updating the robots. There are multiple robots in the world model and vision model. Which visual robot corresponds to which world model robot? Some heuristics to reduce this problem are described next.

The Kalman filter uses variances to represent the confidence of object's location. By assuming the error distribution is Gaussian we can say we are roughly confident the object should be with in a circle of radius 2σ of the estimated location. σ is the standard deviation and 2σ corresponds to the 95% confidence interval for a Gaussian distribution. With this knowledge, we can find which robots to update by overlapping the Visual Model from the vision system with the World Model to find robots that have overlapping confidence regions. This process is illustrated in Figure 4.14. Once robots are matched, the Kalman filters measurement update equations are applied (see 4.6).

In this procedure if more than 1 robot in the vision model is matched to a robot in the world model, then extra robots are created in the world model. These extra robots are copies of the matched world model robot. This procedure ensures a robot in vision model is matched to exactly one world model robot. At the end of the process only the best 4 robot are selected for each team.

What if a vision model robots is matches to 2 world model robots? We can update both the robots using the same vision robot, but it is not logically sound to still keep 2 robots when you only see one robot. But simply removing the 2 robots and replacing with it with 1 robot is also not sound because, the one robot can be temporarily out of sight. To minimise this situation, the robots in the world model under go a merging phase at the very beginning. The condition for matching 2 robots is the same as those in figure 4.14. This way world model robots that are too close will be merged together. After the merge, if the vision system tell you there are two robots not one, then during the update phase described above the robots will be split up again. This method reduces the number of robots needing to be kept. This is under the conception that, when you are looking away from the robots, knowing there is a robot there is enough, the number of robots is less important. The equation for combining the robot is in figure 4.15. In these equations, the final variance of the merged robot is derived from the measurement equation of Kalman filter (see figure 4.6). This can be understood by thinking one robot location is a measurement that came after the other robot location. It can also establishes the idea that if the robots have similar variances, then the merged result should be more certain and hence have lower variance. This equation does exactly that. The code we used the competition actually uses the harmonic mean instead because we didn't come to the consideration of the above conditions. Nevertheless, the equation used in the code looks almost the same except the value obtained using harmonic mean is twice the size.

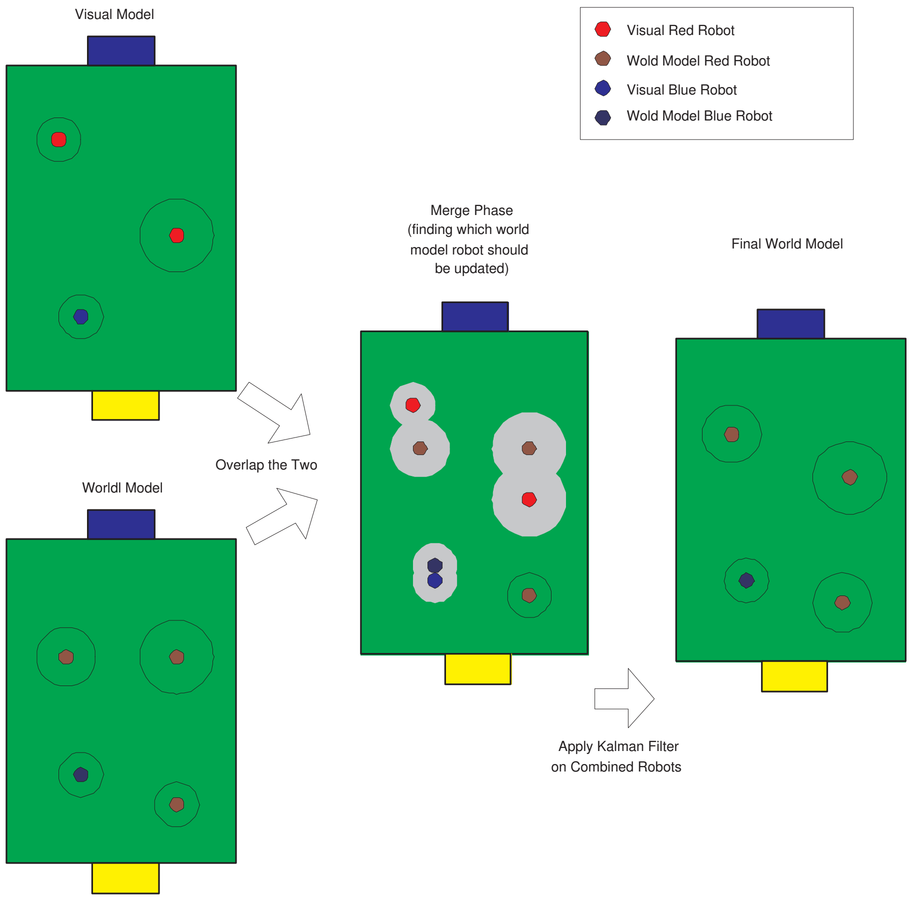


Figure 4.14: Kalman Filter Robot Update

4.7.2 Time Update

To predict the next location of a ball or a robot we must build a world model that records the direction and velocity of the object. Unfortunately, these measurements are difficult to obtain with the current sensors. Therefore, the time update in the Kalman filter algorithm is simulated by having an artificially high movement variance that assumes the object will move in any direction randomly. The ball has higher variance when it is out of sight compared to when it can be seen. This is under the logic to update the ball quickly when it comes back in view. Time update simulation is carried out just before the measurement updates are carried out to ensure they are in sync.

$$result.x = \frac{robot1.x \times robot2.var + robot2.x \times robot1.var}{robot1.var + robot2.var} \tag{4.14}$$

$$result.y = \frac{robot1.y \times robot2.var + robot2.y \times robot1.var}{robot1.var + robot2.var} \tag{4.15}$$

$$result.var = \frac{rebot1.var \times robot2.var}{robot1.var + robot2.var} \tag{4.16}$$

Figure 4.15: Robot Combination Equations

4.8 The Wireless Model

In 2002, a new addition to the competition was the availability of wireless network communication between the robots. This technology allows the robots to communicate between each other using any form of messages. This will give the team a new level of cooperation. They can actively participate in team work and tactic planning. Given these motivations and the strive to keep the robots autonomous, we created the idea to only share information about the world. Although sending tactic planning information is possible, this would make the robots depend on communication for an attack. Due to the poor wireless experiences with other leagues in previous years and, given this is the first year we have used wireless in the four legged league we want to be cautious. By sharing only the world information, it allows robots to make decisions independently. At the same time, it will create a strategy that is robust and able survive wireless breakdowns.

4.8.1 Localisation Differences

In theory, one would think that this is a good idea to share the *World Model* since, for example, a robot that is unable to see the ball may still know its location because a team mate that sees the ball had broadcast its world model. The ideal solution was to integrate the world information from other robots into the receiving robot's own world model. This will make minimal changes to strategy and it will keep the world model complete.

Before this idea was committed, we applied a simple test to compare the world models in 2 robots. In this experiment, we placed 2 stationary robots on the field separated by about 1 metre and staring at the ball placed between them. When the robots are placed on the field, we ensure they are well localised by allowing them to pan their head to search for beacons. While the robots are looking at the ball, they broadcast their world model. We visualise the differences of the 2 world models on the wireless world model display by over lapping the 2 world models on the same window. The results were amazing. The world model of the two robots were actually quite different. We expected the world models to be almost identical because we ensured the robots were well localised. From this experiment we found, because of the small differences in error in vision between the robots the resultant world model is also slightly different.

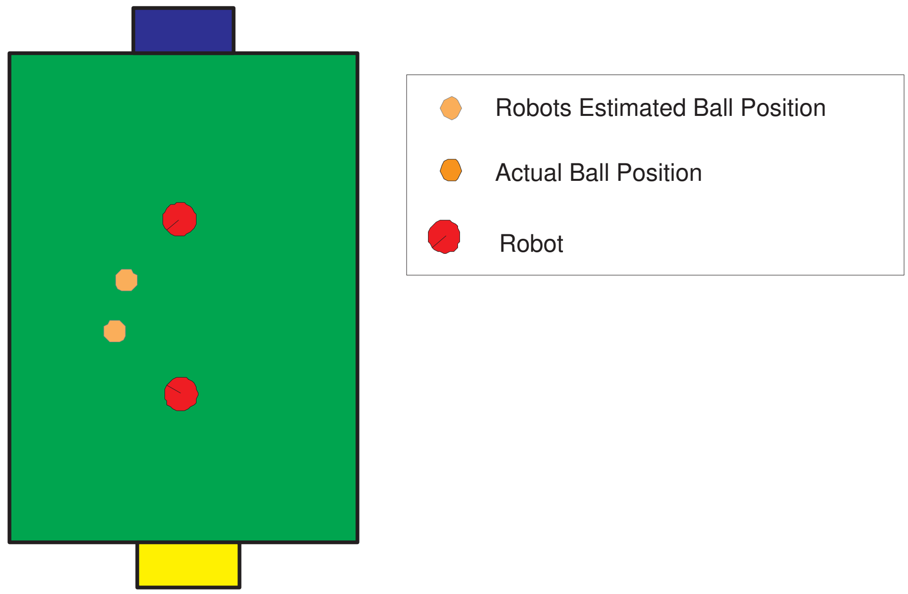


Figure 4.16: World Model Differences Between Robots

This experiment gave us the insight that, if we merge the information from team-mates into the robot's world model it would not be helpful. Imagine the robot is approaching the ball and trying to catch the ball under its snout. This action needs to have accuracy in movement to the centimetre. The robot needs to trust the world model during the approach when the ball is almost under the snout, because the location of a very close ball from the vision module is unreliable. If the information from team-mates were directly incorporated into the world model, then the robot will never be able to catch the ball because the merged information will contain unpredictable differences of other team-mate's world model that will cause the robot to fumble.

4.8.2 Solution, Wireless Model

Considering the differences between robots' world model, a new level of world representation was added –*Wireless Model*. This model is a representation of the world that combines a robot's own world information with those received from team-mates. It is kept separate from the robot's own world model. The variance of an object in the world model received from a team-mate is the sum of the sending team-mate's variance, the object's variance, as believed by the team mate, the age of the information and a small variance for the latency over the wireless network (see figure 4.17).

In calculating the variance of a received object (see figure 4.17), the variable *Age* represents the number of frames we have not heard from the sending robot. It is used in combination with a constant $AGE_VARIANCE$ to increase the variance of the object. The $AGE_VARIANCE$ is the estimated movement of

$$\begin{aligned}NewReceivedObj.var =& ReceivedObj.var + SendingRobot.var + \\ & NETWORK\_VAR + Age \times AGE\_VARIANCE\end{aligned}$$

Figure 4.17: Wireless Object Variance

the object during a camera frame. The constant $NETWORK_VAR$ estimates movement of the objects during the latency of the message from the sender to the robot.

During a game, each robot on the field receives information from every other robot. To combine the world models of all three other robots and its own, we have applied the same technique we used for multiple robot update in world model, shown in Figure 4.14. Except in this case we will be merging the information from 4 world models, one for each team mate. In short, all the world model information are first placed together in a pool. Then, balls and robots of same colour are merged if their variance circle over lap. The radius of the circle is $2\sqrt{var}$ denoting the 95% confidence region of a Gaussian distribution. The equation for merging the information of the object is the same as those shown in figure 4.15. Once the information is merged, the best (lowest variance) 4 robots of each colour and the best ball are stored in the wireless model.

In the wireless model extra information is also stored. These are the sending robot's location and the sending robot's belief of the ball. These information are the unaltered information received via wireless. They are kept to allow the strategy to determine who is the closest to the ball by comparing each robot's belief of their distance to the ball. This is much more accurate than comparing the distance in the resultant wireless model that combines all the world models.

4.9 Evaluation

An evaluation of the localisation method can be done in terms of accuracy, responsiveness and robustness.

4.9.1 Accuracy

It is interesting to compare the old ad hoc method for localisation versus the new method based on the Kalman Filter. One robot was programmed with the old method and another with the new. Both were placed in the same location. The robots constantly panned their heads, searching for landmarks. The behaviour of the old world model is excellent when it is looking at nearby beacons. However, once it is at one end of the field and facing the other end, its estimated position in the world model jumps erratically. This is because of the old world model assumes that beacon distance measurement is always accurate. That is, the error estimate functions are not distant dependent. When the new world model is used, it has similar problems. However, it is much steadier. Thanks to Kalman filters, it is possible to adjust the steadiness of the world model by tuning a few parameters, such as the process noise, Q. Figure 5 shows a comparison of the two world models. We can see that the old world model has a greater scatter of position estimates.

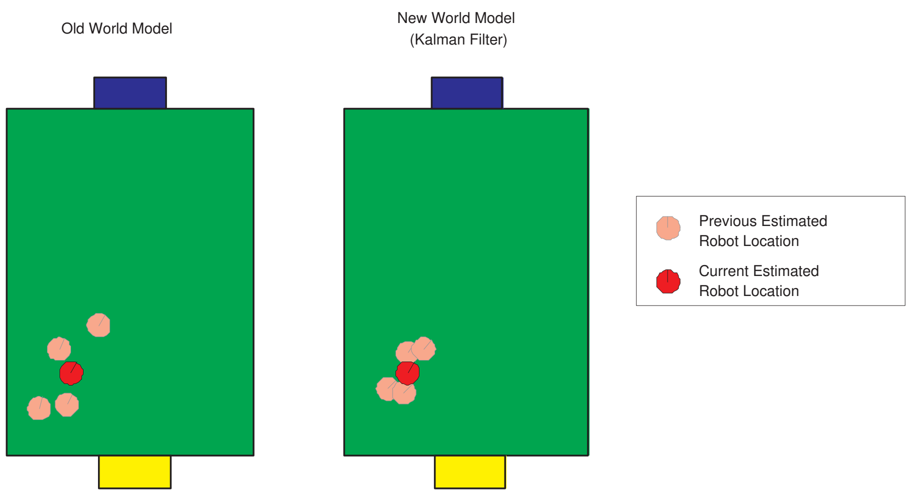


Figure 4.18: World Model Comparison

The introduction of wireless communication in the 2002 competition meant that the robots could share their world models. Theoretically, this means that a robot should be play blind folded, because it can maintain a world model based on the observations of the other robots. In practice, however, it is not so simple. Although, the wireless model incorporates information from all robots, it has the deficiency that because every robot has its own errors in localisation, the resultant wireless model is useful only for coarse-grained behaviours such as getting to a far ball and avoiding interfering with a team mate. In fine-grained behaviours, such as kicking a ball, which needs accuracy to the centimetre, the wireless model is too noisy. To demonstrate this, Figure 6 shows the result of overlapping the world models of two robots. For simplicity, the diagram only contain the ball. The purpose of this demonstration is to show how small errors between the world models of two different robots can greatly affect the final model. Note that both robots have slightly inaccurate estimates of the ball's position. When the estimates are combined, the new ball position is not an improvement.

4.9.2 Responsiveness

Responsiveness is a measure of how fast a robot reacts to new information. That is, how fast is the world model accurately updated? The faster the update, the more reactive is the system. There is a trade off because a robot tat responds too quickly to new information will be seriously affected by noise. Once again, by tuning a few parameters it is possible to adjust for this trade off. This was more difficult with the old world model because the variance of a landmark mainly depended on the percentage of the object seen and not its distance. Observations of these behaviours were made during game play. Wireless communication allowed each robot to transmit its world model to a base station where real-time graphical display enabled us to capture that world model updates. When the beacons were a long way from the robot, they might appear only as tall as 7 or 8 pixels. With a lot of noise, "ghost beacons" could also appear. Under the old world model, the robots would occasionally "teleport" across field then back, causing erratic behaviours. With the new world model, the robots quickly responded to nearby beacons and responded quickly to far beacons only no other landmark had been seen for a long time.

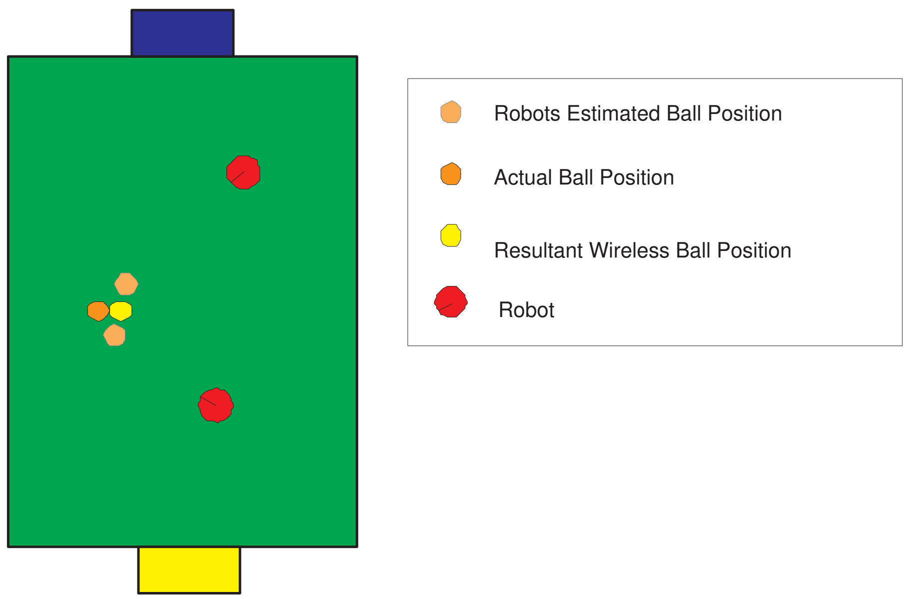


Figure 4.19: World Model Limitations

4.9.3 Robustness

The vision system recognises objects by forming blobs of pixels, all of the same colour. As mentioned previously, distant objects may form blobs containing only a few pixels and thus are difficult to distinguish from noise. A simple remedy is to reject blobs that are smaller than some threshold value. If the threshold is set too high, we risk throwing away useful information. If it is set to low, we risk creating many phantom objects in the world model. This problem was reduced somewhat because we collected statistics on the noise in the data, which could be incorporated into the Kalman filter.

4.10 Conclusion

Despite the many improvements obtained through the use of a principled technique such as the Kalman Filter. Localisation remains a difficult problem. Erratic positioning and frequent errors can be reduced somewhat, but not eliminated. Combining information from other robots can be of use on some occasions, but in others situations, it is better to ignore the information coming from team mates and rely on onboard sensors. For these reasons, the game play software contains many heuristics that attempt to use "common sense" in the application of localisation techniques. Unfortunately, the heuristics are also ad hoc and prone to error. Ideally, one would like to be able to place robots on the field and have them learn to tune parameters and acquire or discard heuristics. When dealing with physical systems, however, one cannot afford to run the thousands of trials often required by learning systems and, so far, learning in simulation has not transferred well to the physical world. Thus, many challenges remain in finding effective methods for dealing with uncertainty and noise in physical systems.

4.11 Future Plans

The localisation module developed has many aspects that can be improved. First, we can extend our Kalman Filter to be multidimensional. This year we have only applied the simplified one dimensional Discrete Kalman Filter into our localisation module to combine information. Next, we can also add methods to estimate the velocity of the object to improve the Kalman Filter. Further more, many of the ad hoc variance constants used this year can be tuned so that they better represent real world situations. This brings a interesting idea. Can we automatically tune those values? Goal localisation is another aspect that can be improved. Attempts have been made this year to solve it but unsuccessful. Finally, the localisation module contains lots of useful information, we should try to use them in the strategy in the future to improve our game play.

4.11.1 Multidimensional Kalman Filter

We have applied a one dimensional Kalman Filter this year. This has placed the assumption that the variance along the x and y axis are the same. When we are localising off one beacon, this is not always true. Since we are more sure about the angle of the object rather than the distance, the variance of the measurement should be spread along the direction of the observed beacon. The ideal variance spread is can be seen in figure 4.20. To have a variance representation that is elongated in a direction we must use Multidimensional Kalman Filter. Having a Kalman Filter that is 2 dimensional, we can describe an object's variance along both x and y axis. This can provide a resultant variance circle that is oval shaped. If we describe the object's variance using 3 dimensional Kalman Filter by having an extra variable then we can indicates the rotation of this oval. Of cause in essence we can have a Kalman Filter to any dimension which can result in any form of variance circle around robot. These ideas can be seen in figure 4.21.

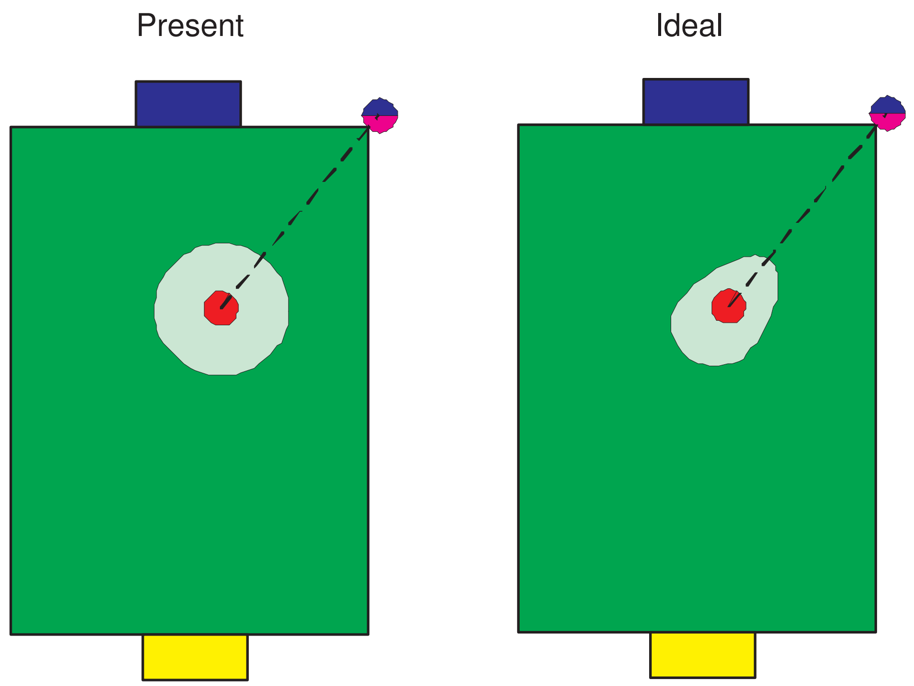


Figure 4.20: Beacon Measurement Variance Comparison

Using a Kalman filter that has higher dimensions we can obtain a more detailed localisation model that has extra information regarding the direction in which the robot is unsure of its location. This can help the strategy to choose the right behaviour. For example, if the robot is in front of the goal, it can be certain about is x value but unsure of its y value, then it may choose to kick the ball along the x direction any way. This just highlights one possibility of using a more complex Kalman Filter. There are implications of having a more complex Kalman Filter. It means there are more constants to tweak and more variables. This implies more difficulty in debugging, testing, and using the resultant localisation information.

4.11.2 Motion Estimation or Object Velocity

Presently we have assumed that we cannot predict the motion of the object. This because we have not built an estimation of its velocity. If this extra feature is created, then the Kalman Filter we applied will be more complete. For example, if we are able to estimate the velocity of the ball, then we can add the prediction into the Kalman Filter process. In result, we can estimate the location the ball is heading even if we loose sigh of it. This will keep the world model more

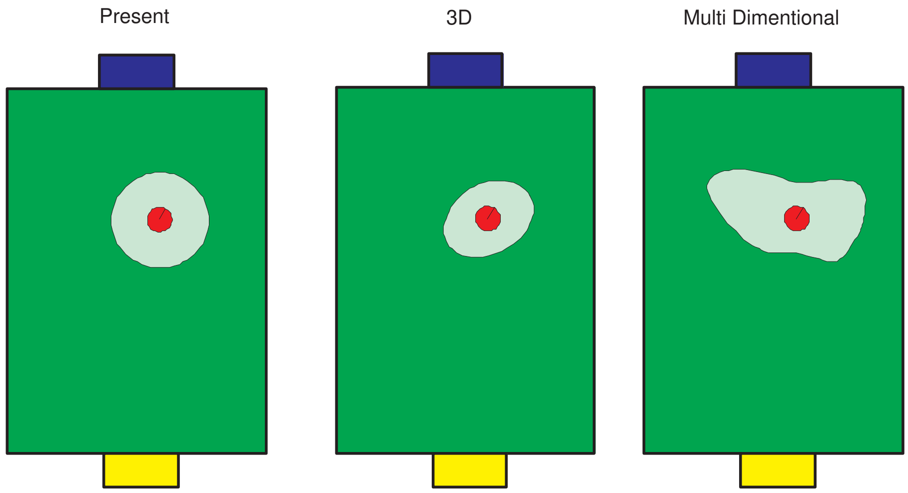


Figure 4.21: Object Variance Comparison

accurate and can aid the strategy by telling it the more likely direction to look for the ball.

4.11.3 (Auto) Variance Tuning

Presently, some variance constants such as the motion of the ball and robots were used. These values are made artificially high because we are unable to provide an estimate of the velocity of the object. They are not the best variance value. They were chosen after a number of tries with different values. The values chosen gave the most stable measurement that can adapt to quick changes.

The values presently chosen are manually entered. They are not always the values that gives the best localisation. An idea is to have the robot walk around and give it a fix object that it needs to model in the world model. The robot will then automatically adjust the constants to best fit the situation. Then we can start moving the object and require the robot to adjust the constants again so that the world model is able follow the motion of the object. Finally, we can remove the object and place it suddenly some where else on the field and require the robot to adjust the constants to adapt to this situation.

This raises another possibility where we can make the robot learn the best variance for the beacons that give it the best localisation. We can make the robot walk around the field and give it the true location on the field, then require the robot to adjust the variance of the beacons.

4.11.4 Goal Localisation

Goal localisation has always been a puzzling problem. This year we have discovered that if we use both the beacon and the goal for localisation, then we can overcome some of the shortcomings of the current localisation procedures. Presently, the robot's localisation is quite poor when it is in front of the gaol and facing it. There is often an underestimation of the distance to the goal. Using this new technique we can accurately tell the location of the robot along the y axis in that situation. Unfortunately, we didn't have time to fully implement it. This can be a goal for next year's team. The details of this localisation technique can be found in the "Localisation" chapter under the section "Goal Localisation".

4.11.5 Using Edges for Localisation

The current localisation techniques are based on the size of the beacon seen in a camera frame. Due to the large size of the field, there is a limited source of localisation information. There are only 6 beacons around the field. If you remember when you (human) play soccer on the field, did you have beacons that tell you where you are located? When we play soccer, we can tell where we are using the markers on the ground. Using this concept, we can localise using the edges of the field and even the edges of the goal.

The carpet of the field is coloured green. Presently, we already identify the colour blob that represents the carpet. If we modify the algorithm to produce a polygon instead of a rectangular bounding box for the carpet then we can use the edge of the polygon to tell us how far we are to the edge of the field or even the direction we are facing on the field. For example if we are on the left side of the field and facing up, we should see a large green polygon with a long left edge on at an angle between 0 and 90 degrees.

Equally we can apply the same concept to goals. The colour blob representing the goal is a much more complex shape. However, understanding that it is a thin C shaped object and painted only on the inside, we can use that observation to decide which side of the goal we are on and possibly the distance to it.

There are many other situations where we can use the edge of objects we observe for localisation. However, this technique requires modifications to the existing blob formation which can make it slow. Also, this technique does not tell you the exact location. For example, if you saw an horizontal edge of the field in front of you, it does not tell your x value. However, if we can introduce a probabilistic model, we can solve this problem by eliminating the different alterative solutions when we obtain more views.

4.11.6 Use all Localisation Information In Strategy

The present strategy does not utilise all the features in the world model and wireless model. It does not fully utilise the capabilities to recognise multiple

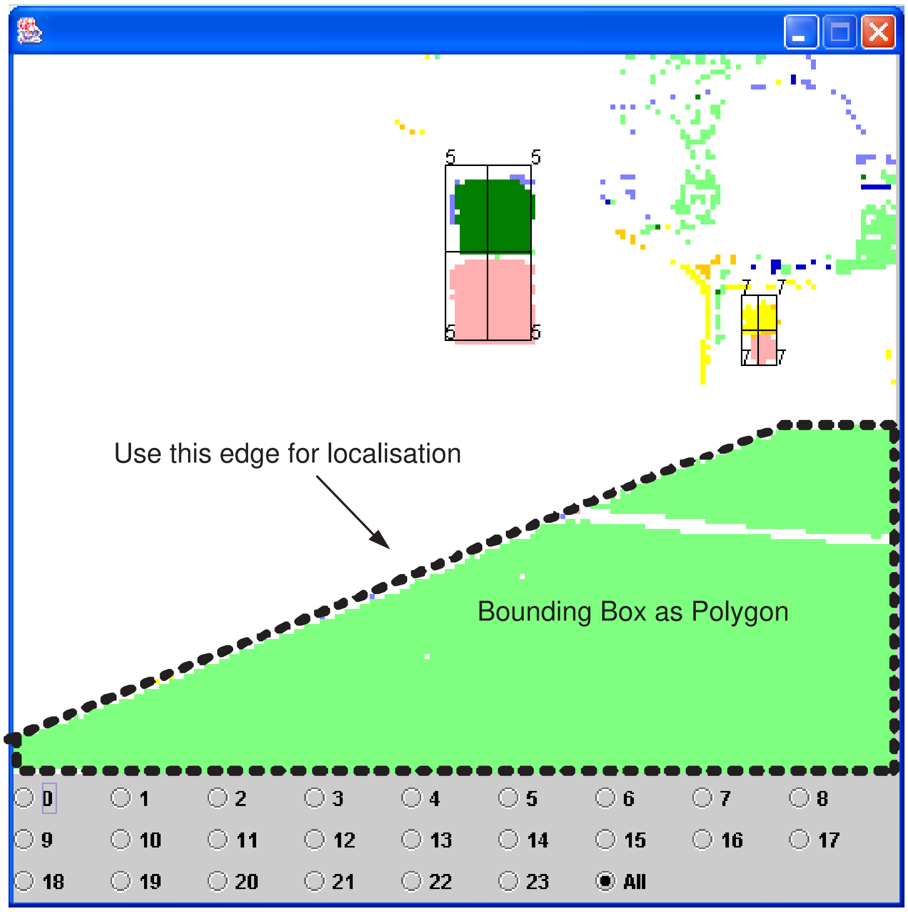


Figure 4.22: Edge Localisation

team-mates for ball passing or recognising multiple opponents for opponent avoidance. The present strategy also does not use the opponent information received in the wireless model. If all these capabilities can be used in the future we can improve strategy planning and team collaboration.

Chapter 5

Locomotion Module

Chapter author: Min Sub Kim.

5.1 Chapter Overview

The locomotion module is responsible for handling the motion of the robot, and can broadly be divided into walking and kicking routines.

This chapter outlines both the walking and kicking routines, and also covers the interface used from outside the locomotion module to drive the routines. It is concluded with a review of related work and a summary of some ideas for future improvements of the module.

5.2 Walking Routines

5.2.1 Section Overview

The walking routines used this year originated from the UNSW 2000 team, and have not changed dramatically since then. In general, the underlying calculations used to generate the robot walk cycles have been preserved since its original implementation.

The walking routines are very flexible, operating on a set of supplied parameters. These parameters can be defined in such a way to produce particular stances and walking styles.

This section consists of a walkthrough of the implementation, a discussion of walk locus and stance variation, and an outline of some customised applications of the walking routines.

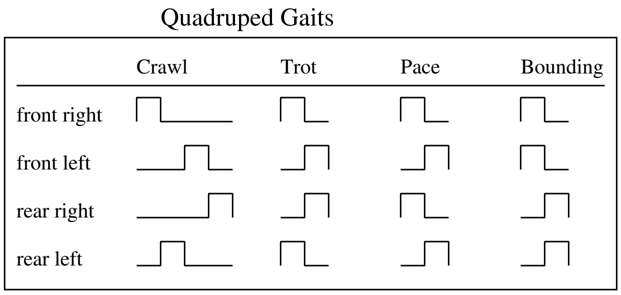


Figure 5.1: A summary of the reviewed quadruped gaits, showing the synchronisation and timing of steps for each leg. This diagram is adapted from [HFT+99].

As a final note, this documentation on the walking routines borrows heavily from the ParaWalk Documentation appendix of the UNSW 2000 team's thesis report. The original author of that section, Bernhard Hengst, is acknowledged and credited for his work in developing the walking routines [HIPS00].

5.2.2 Implementation Walkthrough

Explanation Overview

There are actually three different walking styles implemented in the core of the locomotion module. For simplicity, the simplest walking style will be explained here, followed by a brief outline of the modifications added to the other walking styles.

Gaits

To begin, all of the walking styles used employ the trot gait, in which the diagonally opposite legs are synchronised. This means that as the robot walks, the weight of the body is balanced diagonally across the robot, and shifts with each step taken.

The alternatives considered to the trot gait were the crawl, pace, and bounding gaits. The crawl gait has the robot moving one leg at a time, such that at any point in time, the robot is able to balance on three legs. The result is a slow but steady walk, as the weight of the robot body is balanced on three legs, like a tripod. With the pace gait, the robot moves the legs of one side at a time, that is, the left front and back legs, then the right front and back legs, and so on. In this way, the weight of the robot shifts from side to side. Finally, the bounding gait synchronises the movement of the front legs together and the back legs together. The weight of the robot shifts between the front and the back of the robot body. The pace and bounding gaits both proved ineffective for the Sony robots, while the crawl gait restricts movement too much, making it unsuitable for match play.

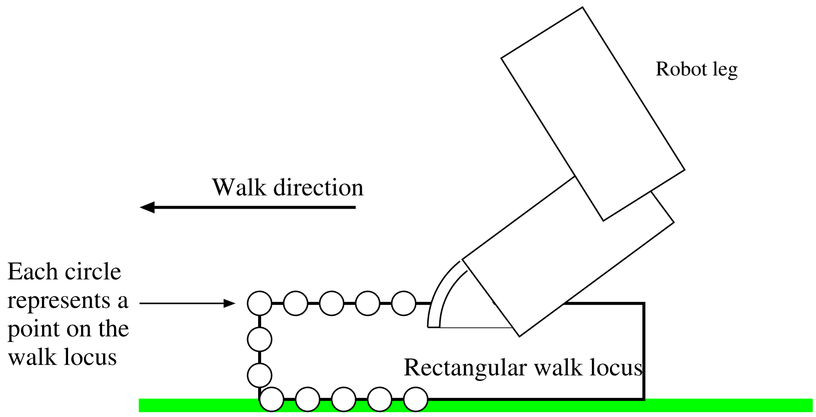


Figure 5.2: The rectangular walk locus.

Walk locus, step cycles, half-steps

With the gait set, and with parameters supplied, it is possible to define the walking motion. In order to do this, it is necessary to define the step cycle and the walk locus. The walk locus is the set of points that the end-point of the robots leg moves along to produce the walking motion, the end-point being the robots paw. A simple way to visualise the walk locus is to consider having the robot body placed on a stand, with the legs moving on their own; the walk locus is the shape that is traced out by the robots paw. A step cycle, then, is defined as a traversal through the walk locus.

To produce a walking motion, the legs must not all be at the same point in the walk locus at the same time. Essentially, the legs must move out of phase of each other in the step cycle. Consider, for example, the way in which humans walk. One leg is lifted, moved forward, and then dropped, while the other stays where it was. Once the first step has been taken, the other leg is then lifted and basically mirrors the same action taken by the first step. In essence, the legs move one step out of phase of each other, and thus alternate between stepping forward and staying still. This is exactly what is done with the Sony robots, and given the trot gait, the legs are taken in diagonally opposite pairs. These pairs then move exactly half a step cycle out of phase of each other. Since each of these steps occupy half of the step cycle, they are called half-steps.

The half-step divides the full step cycle into two halves that serve specific roles. In the simplest walking style, the walk locus has a rectangular shape, and is timed in order to produce one half step that occupies the bottom edge of the rectangle, while the other covers the rest of the walk locus - lifting the leg, moving it forward, and dropping it. The reason for dividing the step-cycle this way is to ensure that at any point while the robot walks, at least one pair of legs is on the bottom edge of the walk locus. This produces a steadier walk, which is helpful in that it keeps the camera of the robot steady, thus making it easier to focus on the ball while moving towards it.

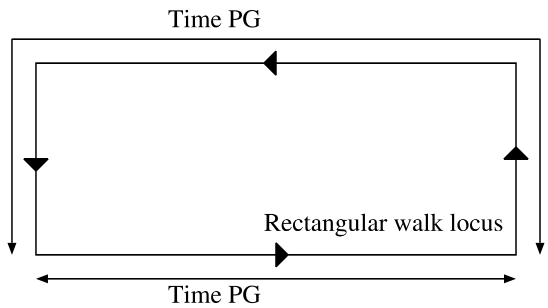


Figure 5.3: The timing of the paw movement around the rectangular walk locus. In short, if it takes time PG to move the paw along the bottom edge of the locus, it also takes time PG to move the paw along the remainder of the locus. Thus, a whole step cycle takes 2PG time.

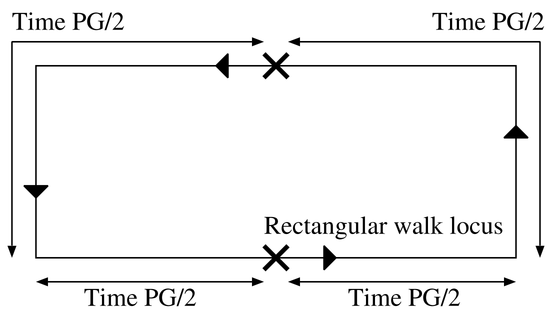


Figure 5.4: The X's denote the points on the rectangular walk locus at which new walk parameters are accepted. Adding the duration around the walk locus, it follows that if the robot is continuously walking, new walk parameters can be given in intervals of duration PG.

Setting parameters

With the above definitions in mind, the implementation can now be explained. Firstly, the input walk parameters are recorded if the robot is at a point in the step cycle where it is able to accept them. These points are when the robot is in the middle of a half-step; that is, either the paw is halfway through the action of stepping forward, or it is halfway through the bottom edge of the walk locus. The reason for restricting the points at which walk parameters can be changed in this manner is to keep the robot better balanced, since changing the direction or stance of the walk at other points may result in erratic motion that leaves the robot less stable. In any case, removing this restriction was tried late in development, but did not produce any noticeable advantages, and was not explored further.

Walk calibration

Once the walk parameters are set, they need to be calibrated in order to produce the intended effect on the field. The reason for this is that the robots have many peculiarities that affect the results that the walk produces on the field. Thus, even though the parameters may, for example, state that the robot will move some distance forward, the robot may not move the intended distance for some reason. To work around this, the input parameters are calibrated, with the calibrated values then used in the remaining calculations. In short, the purpose of parameter calibration is to ensure that the intended effect is produced on field, while hiding whatever values are used in the walk calculations.

The calibration itself was done empirically through measurements and applying curve fitting techniques.

Odometry

Odometry refers to the estimated movement of the robot according to the locomotion module. In other words, if the robot knows where it is initially, and takes a step forward, it should realise in its world model that its position should be approximately a step forward from where it was initially.

While calibrated values are used when calculating the walking motion, the input parameters are fed uncalibrated to the world model. Accurate calibration is essential in ensuring that the movement on field accurately reflects the intended movement. These values are passed immediately after calibration.

There are two main problems with relying on odometry. The first problem is that odometry needs to be very accurate. As the robot moves around, the error margin accumulates if the odometry is not exact for each step, thus leading to a large error margin over time. Furthermore, in match play, the robot will run into obstacles, such as opposition robots and the edges of the field. Odometry cannot be relied upon in this case since it is physically blocked, and thus will not produce the intended movement on field.

Generating the walk cycle

At this point, all of the initial calculations are set and the walk motion can be generated. In short, this is done by calculating the next point on the walk locus, and then applying inverse kinematics to determine the angles that the robot joints need to move to, and repeating the process for each leg, and finally repeating the process for the next few points of the step cycle.

The next point on the walk locus for a particular leg is calculated by examining the walk parameters, and then plotting the end-points of the locus. With the locus positions set, calculation of the next point on the walk locus is simply done by deciding how much of the current step cycle the robot has completed, and then picking a point on the locus based on that.

Once the paw position has been set, inverse kinematics are applied. Inverse kinematics finds the angles at which the leg joints need to be set in order for the paws to reach a particular end-point. In this case, the end-point is the next point on the walk locus.

Finally, with the joint angles calculated, the values are fed into a buffer to dictate movement for the next camera frame. Since the next few points of the step cycle are calculated in each camera frame, the result is smooth motion of the robot legs.

Variations on the simple walk

The first variation on the simple walk is the canter walk. In this style, a canter motion is introduced by moving the robot body in a sinusoidal wave pattern. This produces a rocking motion in the walk, and improves the speed of the walk considerably.

The canter walk is quite stable, though it shakes the camera more than the regular walk. Nevertheless, this version of the walk is used as the standard walk in match play as it is significantly faster than the regular walk, but still steady enough for most purposes.

The second variation is the trapezoidal walk, named after the walk locus shape employed, that is, a trapezium, where the top of the locus is longer than the bottom.

The rationale behind the trapezoidal walk is to avoid dragging the robots claw on the ground as it moves forward. The details of this walking style are covered in detail in the walk locus variation section.

5.2.3 Geometry Outline

The geometry of the locomotion module translates the walk parameters into the motor angles of for each leg joint. This can be divided into three steps:

- Calculating the locus end-point relative to the ground
- Calculating the locus end-point relative to the robot shoulder/hip
- Calculating the leg joint angles

An explanation of each process follows.

Calculating the locus end-point relative to the ground

The first step needs to take the given forward, sideways, and turn parameters and "plant" the walk locus. This is easily done by considering a top-down view

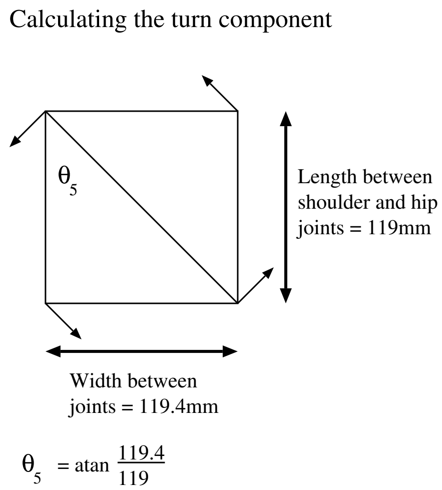


Figure 5.5: The calculations for the turn component. The forward and sideways components can be directly translated to vectors, but the turn component is different for each leg.

of the robot and using vector addition with the supplied movement parameters for each leg.

The locus end-points calculated here are in the f-s (forward-sideways) coordinate system. These points are relative to the point on the ground directly below the robot shoulder or hip.

For the front left leg:

- $fstart = FC - TC \sin\theta_5 + ffo$
- $sstart = SC + TC \cos\theta_5 + fso$

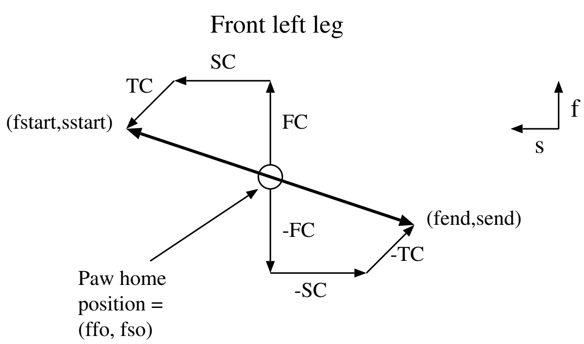


Figure 5.6: The calculations for the front left leg. FC is the forward movement component, SC is the sideways movement component, and TC is the turning movement component.

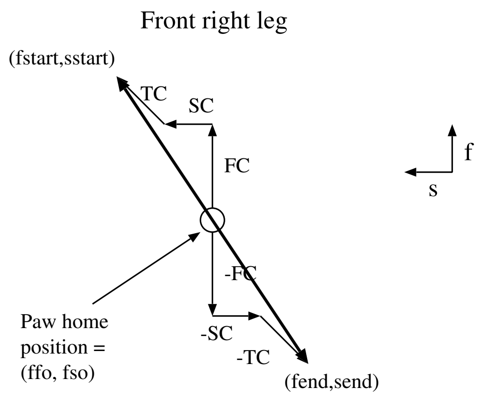


Figure 5.7: The calculations for the front right leg. FC is the forward movement component, SC is the sideways movement component, and TC is the turning movement component.

- $fend = -FC + TC \sin\theta_5 + ffo$
- $send = -SC - TC \cos\theta_5 + fso$

For the front right leg:

- $fstart = FC + TC \sin\theta_5 + ffo$
- $sstart = SC + TC \cos\theta_5 + fso$
- $fend = -FC - TC \sin\theta_5 + ffo$
- $send = -SC - TC \cos\theta_5 + fso$

However, since the angle values are reversed between the left and right legs for moving sideways, *sstart* and *send* need to be adjusted:

- $sstart = SC + TC \cos\theta_5 + fso$
- $send = -SC - TC \cos\theta_5 + fso$

For the rear left leg:

- $fstart = FC - TC \sin\theta_5 + bfo$
- $sstart = SC - TC \cos\theta_5 + bso$
- $fend = -FC + TC \sin\theta_5 + bfo$
- $send = -SC + TC \cos\theta_5 + bso$

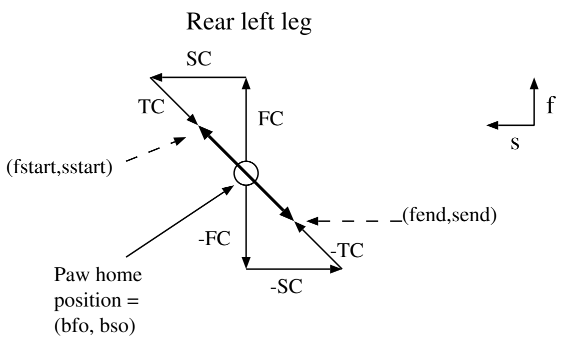


Figure 5.8: The calculations for the rear left leg. FC is the forward movement component, SC is the sideways movement component, and TC is the turning movement component.

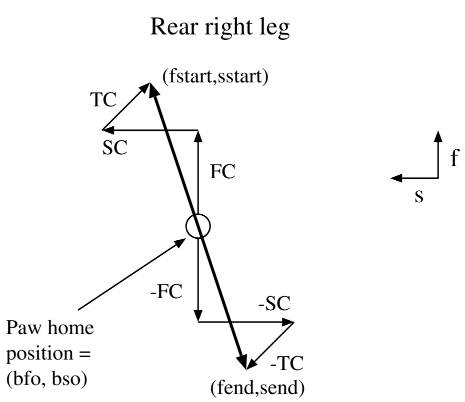


Figure 5.9: The calculations for the front right leg. FC is the forward movement component, SC is the sideways movement component, and TC is the turning movement component.

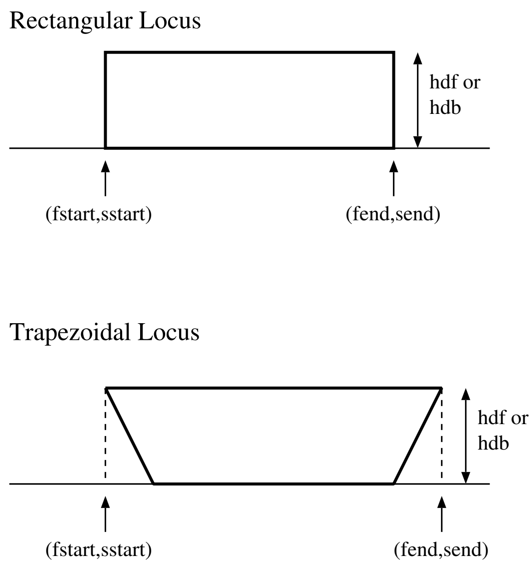


Figure 5.10: Creating the walk locus from the points $(fstart, sstart)$ and $(fend, send)$, and the leg lift parameter, hdf or hdb.

For the rear right leg:

- $fstart = FC + TC\sin\theta_5 + bfo$
- $sstart = SC - TC\cos\theta_5 + bso$
- $fend = -FC - TC\sin\theta_5 + bfo$
- $send = -SC + TC\cos\theta_5 + bso$

However, since the angle values are reversed between the left and right legs for moving sideways, $sstart$ and $send$ need to be adjusted:

- $sstart = SC - TC\cos\theta_5 + bso$
- $send = SC - TC\cos\theta_5 + bso$

The points $(fstart, start)$ and $(fend, send)$ thus form the limits of the walk locus on the f-s coordinate system. The f-s coordinate system is actually extended into the f-s-h (forward-sideways-height) coordinate system in implementation. This is because the f-s system is sufficient to cover the ground-based points on the walk locus, but insufficient to cover the points that are not ground-based. Thus, the additional dimension h is introduced, to describe points where the robot paw is lifted off the ground. h is actually calculated from the robot shoulder or hip, for convenience in the next step.

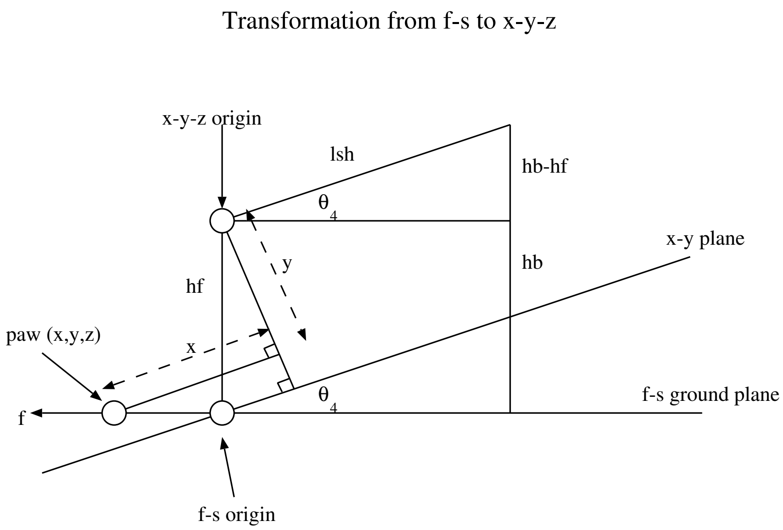


Figure 5.11: Transformation from the f-s coordinate system to the x-y-z coordinate system.

Calculating the locus end-point relative to the robot shoulder/hip

The second step of geometry takes the points calculated in the f-s-h coordinate system and translates them to the x-y-z coordinate system. The x-y-z coordinate system is relative to the robot shoulder or hip, whichever is relevant for the concerned leg. It is effectively the same as the f-s-h system, except that the x and y directions are adjusted for the slant of the robot body. Thus, the x direction points straight ahead and parallel with the robot body, while the y direction points down and is perpendicular to the robot body.

The transformation is then calculated by applying a projection of x-y-z on the ground plane. s is directly transformed to z, while x and y are calculated using trigonometry based on the angle of the robot body (refer to figure 5.11).

The transformation is thus:

- $z = s$
- $y = h \cos\theta_4 - f \sin\theta_4$
- $z = h \sin\theta_4 - f \cos\theta_4$

where h is calculated according to the current position on the walk locus for each leg.

As a note, since the y direction points downwards, the previous calculation of h from the robot shoulder or hip can be directly used here without transforming it.

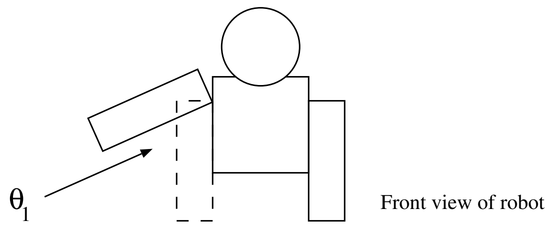


Figure 5.12: A front view of the robot, showing θ_1.

Calculating the leg joint angles

Inverse kinematics are finally applied in this step to convert the points in the x-y-z space to leg joint angles. However, before this can be done, the last coordinate system needs to be introduced. This is the u-v plane, which is also called the limb plane (note that this unrelated to the y-u-v colour space in colour detection). The u-v plane has its origin at the robot shoulder or hip, and it is the plane that includes the entire robot leg. The v direction is parallel to the upper leg, pointing towards the knee joint, while the u direction is simply perpendicular to v.

To visualise the u-v plane, consider attaching a square of cardboard, flat on the side of the upper leg. The limb plane thus rotates around the x direction if the leg is raised sideways, and it rotates around the z direction if the leg is raised forwards. Bending the knee does not affect the limb plane.

There are three joints in each leg, and thus it is necessary to calculate the angles for each of them to send to the effectors. These three joints will be referred to here as θ_1, θ_2, and θ_3, with the following meanings:

- θ_1 = the rotation of the upper leg around the z direction, or the angle at which the limb plane intersects the body side (this is called the "shoulder" value in the kick macros).
- θ_2 = the rotation of the upper leg around the x direction, or the angle at which the upper leg deviates from the angle of the robot body in the x-y plane (this is called the "joint" value in the kick macros).
- θ_3 = the angle at which the knees are bent (this is called the "knee" value in the kick macros).

θ_3 is calculated first. Because the knee is hinged on one side, the calculations are not completely straightforward.

From figure 5.15,

$$\theta_3 = 2\pi - \theta_A - \theta_B - \theta_C$$

Then:

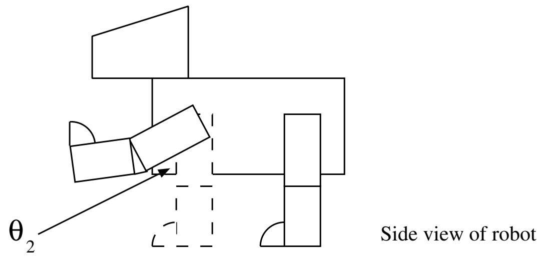


Figure 5.13: A side view of the robot, showing θ_2.

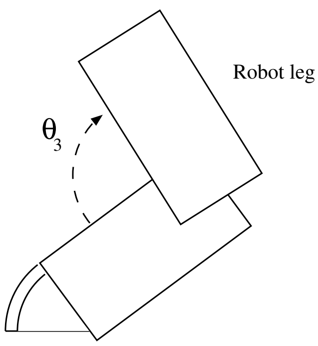


Figure 5.14: A close view of the robot leg, showing θ_3.

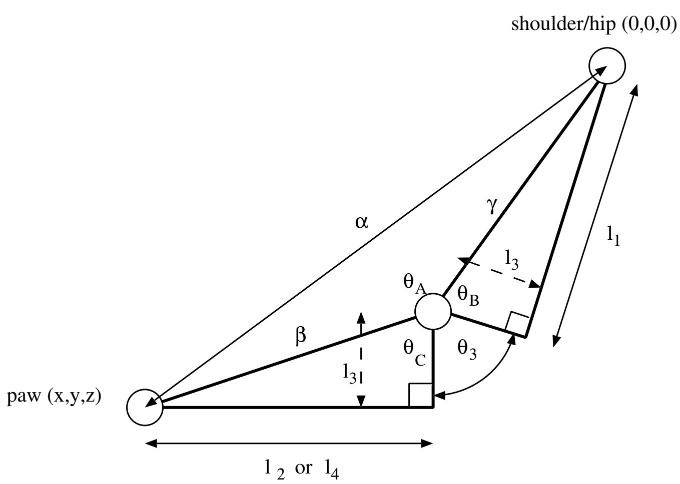


Figure 5.15: Inverse kinematics for θ_3.

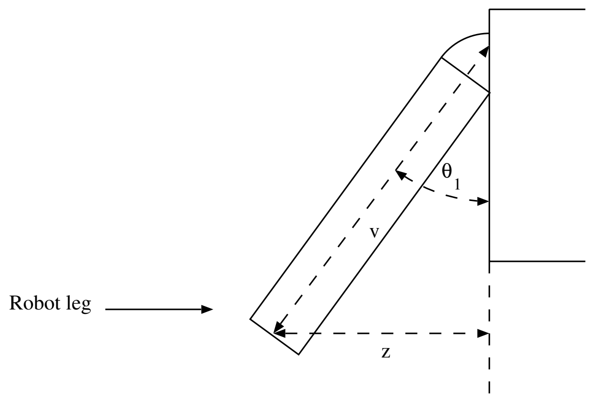


Figure 5.16: The relationship between $theta_1$, v, and z.

$$\cos\theta_A = \frac{\beta^2 + \gamma^2 - \alpha^2}{2\beta\gamma}$$

where

$$\begin{aligned}
\alpha &= \sqrt{x_2 + y^2 + z^2} \\
\beta &= \sqrt{{l_2}^2 + {l_3}^2} \\
\gamma &= \sqrt{{l_1}^2 + {l_3}^2}
\end{aligned}$$

Then:

$$\begin{aligned}
\theta_B &= \arctan(l_1/l_3) \\
\theta_C &= \arctan(l_2/l_3)
\end{aligned}$$

The values of l_1, l_2, l_3, and l_4 are fixed measurements, specified in the locomotion module.

θ_1 is calculated next. To begin, from figure 5.16, it is observed that:

$$\sin\theta_1 = \frac{z}{v}$$

and thus

$$\theta_1 = \arcsin\frac{z}{v}$$

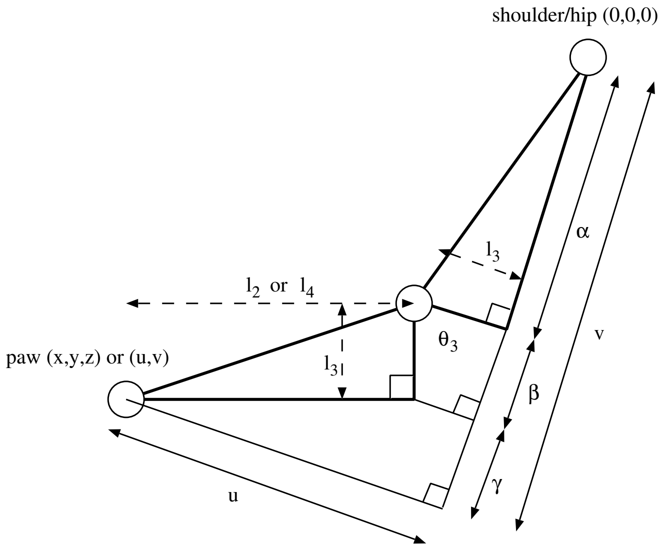


Figure 5.17: Calculating v by summation.

Knowing the paw z value, the only thing left to find is the paw v value. From figure 5.17, this can be seen as:

$$v = \alpha + \beta + \gamma$$

Thus, v is calculated piece-wise as such. To begin, it is clear that:

$$\alpha = l_1$$

Next, for β, observe from figure 5.18, that:

$$\beta = \delta \cos \theta_C$$

The triangle enclosing θ_3, θ_A, and θ_B is a similar triangle, and so

$$\theta_A = \theta_B = \frac{\pi - \theta_3}{2}$$

Thus $\theta_C = \frac{\theta_3}{2}$, and δ is calculated as:

$$\delta = 2l_3 \sin \frac{\theta_3}{2}$$

Finally, this means:

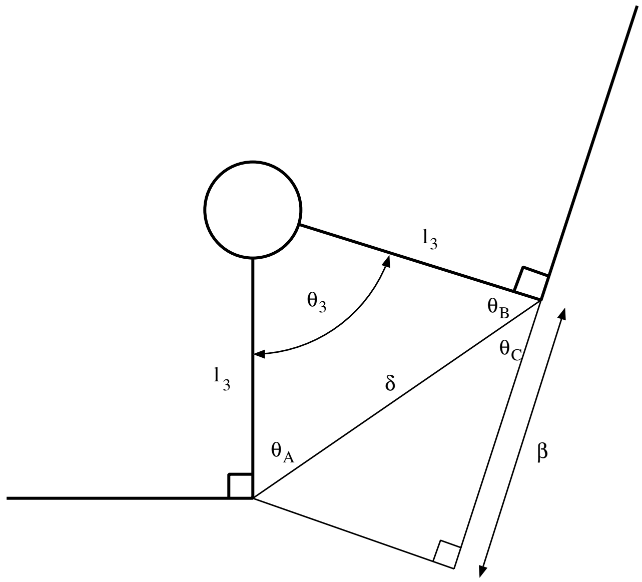


Figure 5.18: Calculating β.

$$\beta = 2l_3 \sin\frac{\theta_3}{2}\cos\frac{\theta_3}{2}$$

Lastly, calculating γ, from figure 5.19, comes out as:

$$\gamma = l_2 \cos\theta_D$$

Using the previous calculations and the angle of revolution, it is found that $\theta_D = \theta_3$, and thus $\gamma = l_2 \cos\theta_D$.

Finally, this allows the value of θ_1 using the summation that gives v, thus giving the final formula:

$$\theta_1 = \arcsin\frac{z}{l_1 + 2l_3 \sin\frac{\theta_3}{2}\cos\frac{\theta_3}{2} + l_2\cos\theta_3}$$

θ_2 is calculated last. Because the final result of the calculations can be interpreted as either positive or negative, a check is made before deciding which way to interpret it.

From figure 5.19, it can be derived that:

$$u = l_2 \sin\theta_3 + 2l_3 \sin^2\frac{\theta_3}{2}$$

Then, taking the projection on the x-y plane gives:

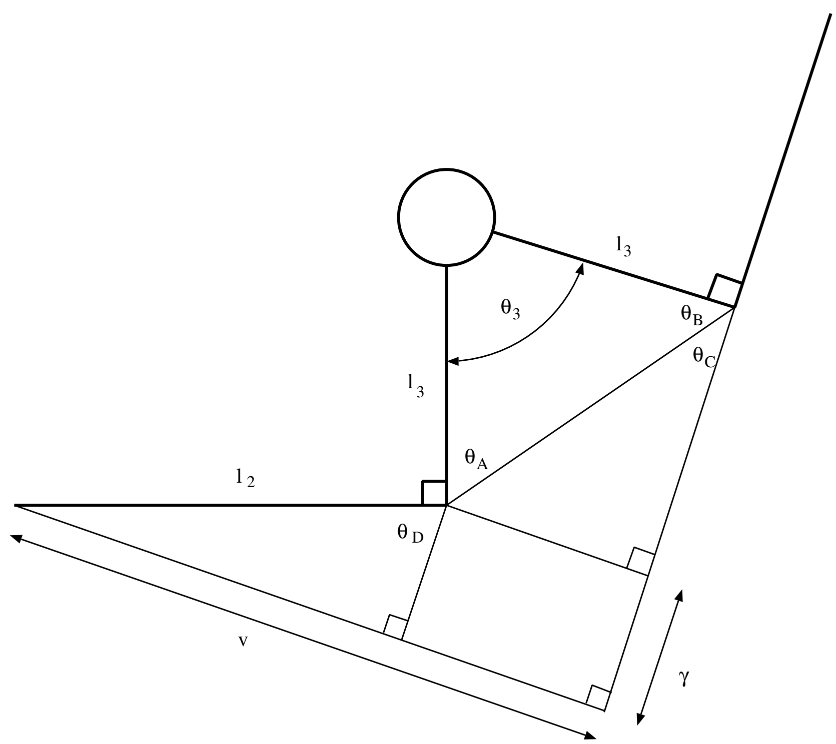


Figure 5.19: Calculating γ, and u.

$$\begin{aligned} x &= v\cos\theta_1 \sin\theta_2 + u\cos\theta_2 \\ y &= v\cos\theta_1 \cos\theta_2 - u\sin\theta_2 \end{aligned}$$

Then, solving by substitution:

$$\sin\theta_2 = \frac{x - u\cos\theta_2}{v\cos\theta_1}$$

Therefore:

$$\begin{aligned} y &= v\cos\theta_1\cos\theta_2 - u(\frac{x - u\cos\theta_2}{v\cos\theta_1}) \\ yv\cos\theta_1 &= v^2\cos^2\theta_1\cos\theta_2 - ux + u^2\cos\theta_2 \\ yv\cos\theta_1 + ux &= \cos\theta_2(v^2\cos^2\theta_1 + u^2) \\ \cos\theta_2 &= \frac{yv\cos\theta_1 + ux}{v^2\cos^2\theta_1 + u^2} \\ \theta_2 &= \pm\arccos\frac{yv\cos\theta_1 + ux}{v^2\cos^2\theta_1 + u^2} \end{aligned}$$

5.2.4 Walk Locus Variation

The walk locus was the subject of some experimentation, with most variations focusing on changing the locus shape. Most of these variations did not improve the walk, with the only notable improvement coming with the trapezoidal walk.

Initial experiments

Initially, three variations were tried on the existing walk. Two of these changed the shape of the locus, and the other changed the timing of the movement around the walk locus.

The first locus shape variation changed the locus into a half-elliptical shape. The bottom half of the walk locus is flat, with the top half taking an elliptical curve. The reasoning was that the curve around the upper half of the walk locus would result in a smoother stepping action than the rectangular locus, thus producing a more natural walking motion. However, this locus change turned out to be far less effective than the rectangular locus. The reason for this is that the claws of the rear paws would drag on the field when walking. This greatly reduced the effectiveness of the walk, as the robot was always stuck trying to move forward, and thus the half-elliptical locus was abandoned.

The second locus shape variation tried was a trapezium, with the lower edge of the trapezium being longer than the upper edge. Like the half-elliptical locus,

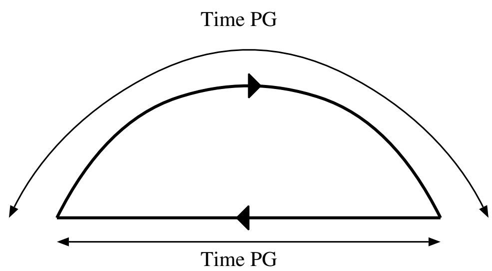


Figure 5.20: The half-elliptical locus.

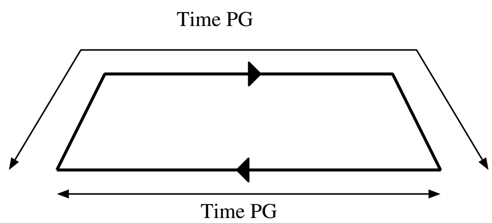


Figure 5.21: A trapezoidal locus with the bottom edge longer than the top edge. Note that this is not the locus used in the faster trapezoidal walk.

this focused on trying to produce smoother steps, but also failed for the same reason as the half-elliptical locus. In general, it seemed that the claws of the rear paws presented a real hindrance to walking forwards quickly, as they would drag on the field and reduce the robots momentum.

The last experiment tried was to change the timing of the step cycle. With the original rectangular locus and the variations tried above, the timing of the steps is such that the time taken for the paw to move along the bottom edge of the locus is equal to the time taken to move around the rest of the locus. This is to try to keep the robot more steady during the walk, as this timing ensures that theoretically, at any point in time, at least two paws are touching the ground. In this variation, the timing around the rectangular locus was changed to a uniform timing, such that the speed of the movement around the walk locus was uniform for any point. This variation had no serious disadvantages against the existing walk, other than the camera shaking a little more. However, it also did not have any significant advantages over the existing walk, and was not explored further.

Trapezoidal Walk (aka "Zoidal Walk")

One of the dominant problems found with varying the locus shape was that the claws on the rear paws would drag on the field, rendering the walk ineffective. The regular walk with the rectangular locus generally avoids this problem, as the paw is lifted directly up without trying to move it forward, thus avoiding getting the claw stuck. However, the problem is that due to the robots forward

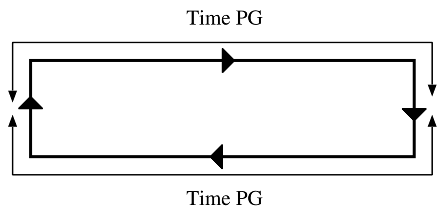


Figure 5.22: The rectangular locus with uniform timing.

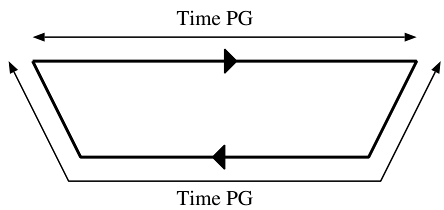


Figure 5.23: The timing around the trapezoidal locus.

momentum, the claw can still be moving forward a little, thus slowing the robot down a little. The trapezoidal walk is aimed at countering this problem.

The trapezoidal shape itself is smaller than its rectangular equivalent. That is, for a given set of parameters, the trapezoidal locus traces out less area than the rectangular locus. The top edge of the two loci are the same, but the trapezoid is shorter along the bottom edge. This is does not mean that the trapezoidal walk cannot take long steps though, since the length of the locus can be stretched via the forward, left, and turn parameters. The timing of the walk cycles is also different, as the timing of the trapezoidal locus is upside-down compared to the rectangular locus. That is, the duration covered by the top edge of the trapezoidal locus is the same as that for the time taken to drop the paw, to move it back, and to lift it. This timing means that there are times in the walk cycle when there are no paws on the bottom edge of the walk locus, producing a bouncing effect.

The primary advantage of the trapezoidal walk is straight walking speed. By

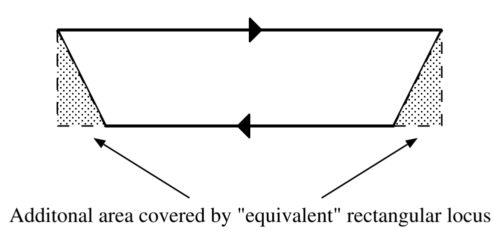


Figure 5.24: A comparison between the size of the trapezoidal and the rectangular locus. The "equivalent" rectangular locus refers to the rectangular locus generated using the same walk parameters.

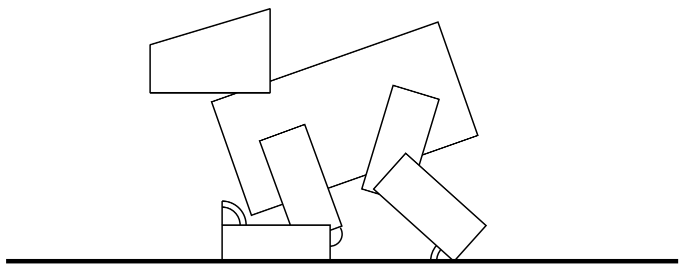


Figure 5.25: The preferred low stance.

significantly reducing the drag caused by the robot claw, the trapezoidal walk allows the robot to maintain momentum and move quickly. In addition, combined with the low stance, the trapezoidal walk is also stable, making it good for charging.

The disadvantages of the trapezoidal walk are that the camera tends to shake more, and it is somewhat less controllable than the regular walk. The trapezoidal walk has a steady walking rhythm that bounces the robot more than the regular walk, causing the camera to shake more. This makes it less suitable when approaching the ball at a close range, since the robot is more likely to lose sight of the ball due to the camera shaking. It is also less controllable in that its movements are more restricted than the regular walk. For example, it has more difficulty when turning or walking backwards.

5.2.5 Stance Variation

When the walking routines were originally developed in 2000, it was found that a low stance with the robot taking long strides was a very effective way of walking. The robot would have the front forelegs flat on the ground when walking, while the rear legs walked "normally" on paws. The low stance was very stable, allowing the robot to aggressively contest the ball against opposition robots, while the low stance allowed the robots to move quickly.

Last year, the stance was re-calibrated for the new robot models, though it stayed mostly the same.

Although the stance stayed the same again for this year, there was some experimentation on trying other stances. Particularly, it was noted that at the competition last year, the French team (Laboratoire de Robotique de Paris) chose a high stance that was faster for straight walking than the low stance. The high stance produced a steady trotting rhythm that appeared to maintain momentum well.

This high stance was closely replicated using the locomotion infrastructure, by simply modifying the walk parameters. However, the copy did not seem as fast as the original, implying that there were other elements that were not correctly reproduced.

Although the high stance does allow for faster forward walking speed, it has significant disadvantages. Firstly, it does not seem to be as stable as the low stance, forcing the robots to play less aggressively or risk being pushed over by other robots. Secondly, it is not as maneuverable as the low stance. The low stance is very flexible since it can reliably combine forward, sideways, and turning movement components. The higher stance has considerably more trouble when trying to combine movement components, particularly when trying to walk sideways. These disadvantages are an expensive trade-off for the small gain in forward walking speed.

5.2.6 Applications of the Walking Routines

The walking routines can be applied to other purposes in strategy. Two examples of this in strategy are the paw kick and the turn kick.

The paw kick is a dribbling action that lines up the ball with the robot paw to hit it with the walking motion. Since the low stance has the robot walking on its front forelegs, the front paws face forward when walking. Combined with the steady walk locus shapes (or more specifically, since the top edge of the walk locus shapes is straight), this predictably hits the ball forward when lined up correctly, and does not require any special locomotion infrastructure modifications.

The turn kick is a kicking skill that allows the robot to turn and hit the ball in one motion. This is simply executed by modifying the walk parameters a little and then turning the robot on the spot. The walking motion will then hit the ball with the front legs, producing a powerful kicking action. Again, this does not require any special infrastructure modifications.

Other examples include skills such as trapping the ball, turning with the ball, and the goalie block. These are executed simply as walks with specialised parameters that move the limbs in the way required.

In summary, the parameterisation of the walking routines allows great flexibility, and has led to simpler development of other skills.

5.3 Kicking Actions

5.3.1 Section Overview

While the walking routines are primarily driven by the input parameters with few hard-coded aspects, the kicking actions, in contrast, are almost entirely hard-coded and have minimal interaction with the input parameters. This is because the kicking actions do not require the flexibility that the walking routines possess.

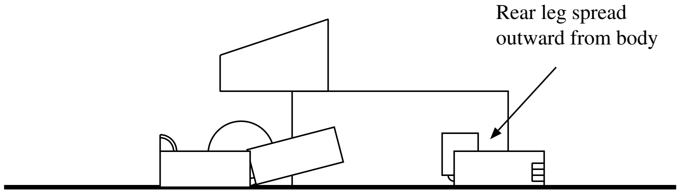


Figure 5.26: The initial step of the forward kick.

This section outlines all the locomotion level kicking actions that were tried in this year's development, as well as other actions that were considered but never implemented.

5.3.2 Forward Kick

Background

The old forward kick originated from the UNSW team in the 2000 competition. The kick is based on a scissors lever action, by dropping the front legs of the robot on the ball in order to shoot it forwards. This idea has served as the basis of many kicking actions since used in the competition.

With the robot model change last year, the forward kick was reinvented for the new robots late in development, and was not heavily integrated into strategy. However, even with the minimal development time, it was still tuned to be powerful enough to shoot the ball across the full length of the old field.

Description

The forward kick is comprised of three actions, and has a duration of 24 camera frames. The time taken to execute the entire action is equivalent to the time taken by the robots to take three half-steps under normal circumstances.

To begin, the robot drops to the ground by reaching forwards and slightly inwards with its front legs, and spreading its rear legs, with the knees bent. While reaching forwards with the front legs may seem an awkward way to begin the kick, if the ball is properly trapped up against the robots chest as described in the skills section, this will have the effect of trapping the ball with the front legs, ready to be hit forward. The head aims straight ahead in this step, which lasts for 6 camera frames.

The robot then raises its front legs over the ball, while lowering its head slightly to hold the ball down. This effectively places the ball in a good position to drop the front legs on, so as to shoot the ball. This step lasts for 12 camera frames.

Finally, the robot drops the front legs onto the ball, while raising its head back up. This shoots the ball out forward a fair distance, able to cover approximately

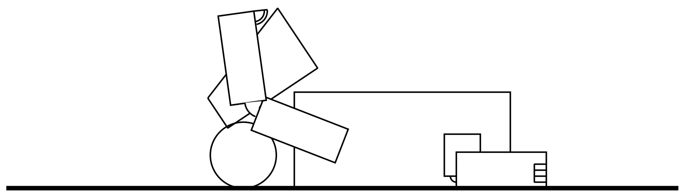


Figure 5.27: The second step of the forward kick.

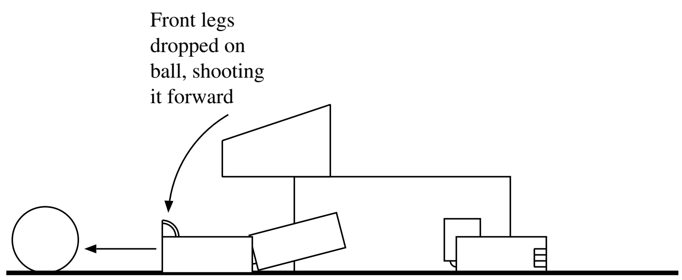


Figure 5.28: The final step fo the forward kick.

half of the length of the new field. Also, the kick is reasonably accurate over short distances, though the accuracy is considerably lower near its maximum range due to the inconsistencies of the field and the ball.

There are two main issues with this kick. Firstly, the reliability of trying to hold the ball down with the head before dropping the front legs is questionable. The ball can sometimes roll out of position in this step, usually due to the robot head hitting it. Secondly, the reliability of the scissors lever action hinges largely on the straightness of the front knees as they are dropped. If the knees are bent fully back, the result is a very weak but almost completely reliable kick. If the knees are fully straight, the result is a powerful but generally unreliable kick, as the ball gets trapped between the front legs. Thus, finding the right compromise is a trade-off between effectiveness and reliability, and to complicate matters further, it seems to depend on the field and the ball, and varies between different robots. In general, this issue is one that is prevalent across all derivations of this kick.

Summary

The old forward kick remained in the final code played at the competition, but was not used. In the end, it was superseded by a variety of other kicks, and was no longer required. However, it served as the starting point for developing new kicks and is still a good benchmark to compare against.

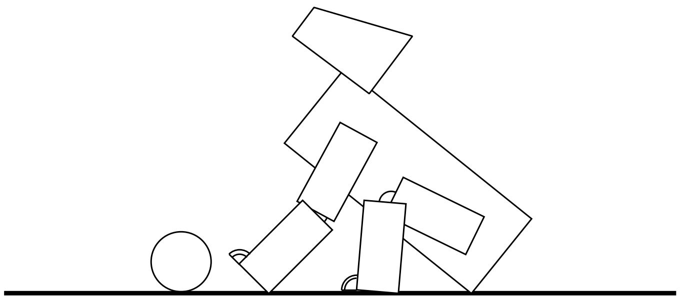


Figure 5.29: The initial step of the headbutt.

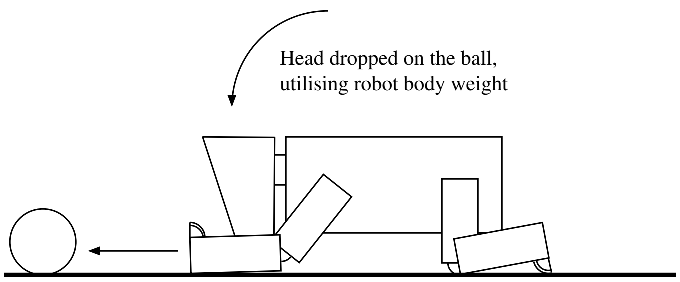


Figure 5.30: The second step of the headbutt.

5.3.3 Headbutts

Background

The UNSW team last year experimented with headbutt actions where the robot would drop its head very quickly on the ball, shooting it outwards. In last year's strategy, the headbutt was used as a kick-off skill for its power.

Description

The headbutt consists of three steps, with a total duration of 30 camera frames. This is a little less than the time taken for the robot to take four half-steps under normal circumstances.

To begin, the robot winds back to prepare the headbutt by kneeling with its rear legs while straightening its front legs. This raises the front of the robot body significantly, placing the body at an angle. The head looks straight ahead relative to the body. This first step takes 10 camera frames.

The front of the robot body is then brought down by moving the front legs forward and bending the knees a little, while the rear legs shift slightly forward. The result is that the robot body will fall until the front legs reach the ground, at which point the robot body is relatively flat to the ground. Finally, to give

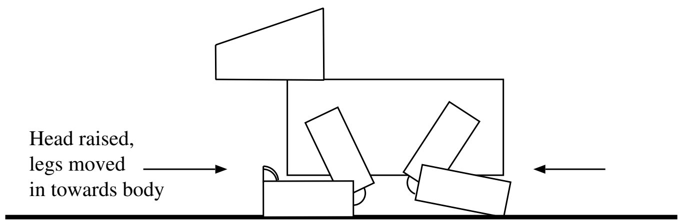


Figure 5.31: The last step of the headbutt.

the ball a good hit, the head is brought down at the same time, such that it points straight down once this step finishes, which has a duration of 10 camera frames.

The last step in the action is a recovery step, to put the robot in a position where it can easily resume walking without losing balance. To do this, the head is raised back up to look straight ahead, while both the front and rear legs are brought back in closer to the body.

Summary

The headbutt is very powerful, more than the other kicking actions described here. However, its lengthy duration and unreliability make it unsuitable for general match play. To execute correctly, the robot needs to be precisely positioned a short distance in front of the ball, and even then the accuracy of the headbutt is poor.

Its use in the strategy last year was limited to kick-off strategies, where the robot could be manually positioned correctly to execute the headbutt. It was useful in this situation since it could very quickly shoot the ball near the goal, but was otherwise not used.

5.3.4 Chest Push

Background

Another skill introduced by last year's team, the chest push is a useful short range kicking skill. It was left unchanged from last year, and was used heavily in strategy.

Description

The chest push is composed of three steps, and has a duration of 18 camera frames. This duration is equivalent to the time taken for the robot to take two and a quarter half-steps under normal circumstances.

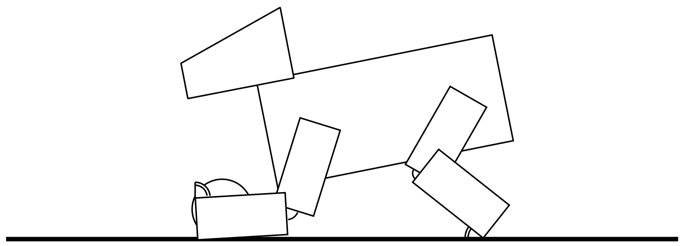


Figure 5.32: The initial step of the chest push.

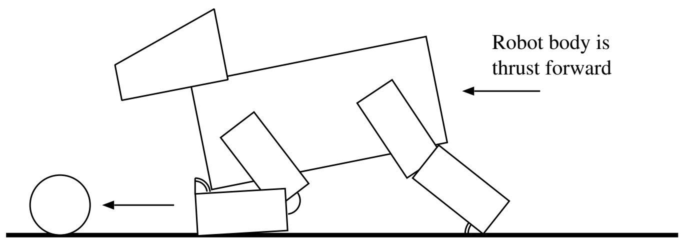


Figure 5.33: The second step of the chest push.

To begin, the robot readies itself by leaning back slightly. This is done by moving the shoulder joints of the robot, so that while the paws stay in place, the body is moved back. It is important to ensure that the robot does not lean too far back, in order to prevent it from falling over or otherwise becoming too unbalanced. This first step lasts 6 camera frames.

With the setup prepared, the robot then thrusts its body forward. Again, this is done by winding back the shoulder joints, thus pushing the robot body forward while the paws stay still on the ground. This is the step that is used to actually hit the ball, as the body acts like a ram to push the ball forward. The extent of the forward push is tuned such that the robot pushes the ball with decent body momentum while ensuring that it does not fall over forwards in the process. This step also lasts 6 camera frames.

The final step simply brings the robot back to the initial ready position, bringing the robot back into a steady position to resume walking.

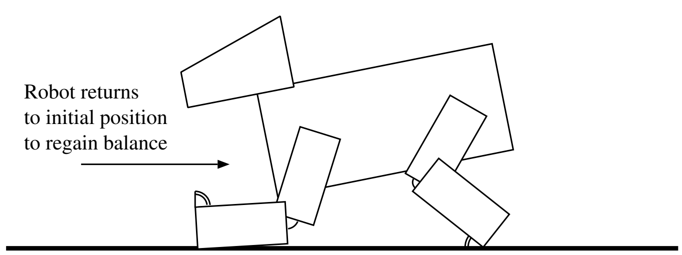


Figure 5.34: The final step of the chest push.

Summary

The primary advantages of the chest push are speed and ease of execution. Its speed helps in pushing the ball quickly past opposition robots, keeping them chasing the ball or just giving them less time to setup their own skills. This is also assisted by the relaxed setup conditions of the chest push, which merely require the ball to be somewhere close in front of the robots chest. In addition, the chest push is very stable, making it useful when trying to push the ball out when crowded by opposition robots.

The main disadvantage of the chest push is its limited range. However, since the use of the chest push in strategy is restricted to short range purposes, this disadvantage never really comes into play.

5.3.5 Aimable Forward Kick

Background

The aimable forward kick is the most useful version of initial developments based on the old forward kick. As its name suggests, this kick can be aimed to a certain extent, providing more flexibility and accuracy.

The development of this kick was originally based on the kicking action used by the Carnegie Mellon team at last year's competition. Their kicking action differed from the forward kick used by the UNSW team in that they made use of the rear legs, standing the robot up at the last second, making their kick slightly more powerful.

Development

As described above, the old forward kick does not make use of the rear legs, as the robot lies flat on its body to execute the kick. Development of this new kicking action began with trying to replicate the action used by the Carnegie Mellon team last year, by modifying the existing old forward kick.

The bulk of the development concentrated on trying to place the rear legs in such a way that they could propel the robot. This proved to be more difficult than expected. Most of the time, the robots legs would try to push forward by winding the hip joint, but the paws would slip on the field, as though the legs were not strong enough to push the robot body weight. Furthermore, it was also difficult in keeping the robot balanced on recovery from the kick. If the robot attempts to immediately resume walking after the kick, the kick must recover in a suitable position to prevent losing balance.

Eventually, through trial and error, the movement of the rear legs was tuned such that they pushed the robot forward during the kick reliably. The movement of the front legs was also adjusted, since the robot body was no longer flat on the ground during the kick. Thus the angles at which the front legs were brought

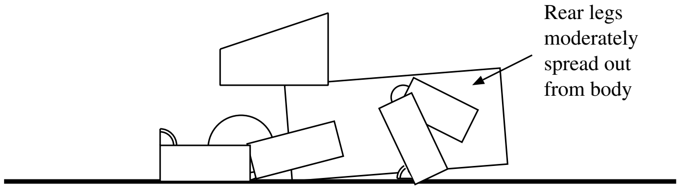


Figure 5.35: The initial step of the aimable forward kick.

down was compensated. The timing of each step was also changed from the old forward kick, extending the initial setup step, so as to give the robot more time to prepare to push forward with the kick.

The new forward kick developed thus did not exactly replicate the Carnegie Mellon kicking action, as their action used the rear legs to raise the back of the robot rather than pushing it forward. However, since the new forward kick seemed sufficiently more powerful and more reliable than the old forward kick anyway, further attempts to replicate the Carnegie Mellon kicking action were not made.

With the general kicking action settled, variations on the symmetry of the rear legs was tried. This led to the aimable forward kick, produced by deliberately dampening the movement of one rear leg. This results in one side of the robot being pushed more forward than the other side, producing a small swaying effect that allows the kick to be aimed a little on execution. This allows for greater flexibility, since it allows a greater margin of error when lining up the target. The extent of the aim was empirically measured to be approximately 15 degrees to either side, allowing a total aim of roughly 30 degrees. The kick is slightly weaker when aimed more to the side, since there is less pushing force generated by the rear legs.

Description

The aimable forward kick consists of three steps, and has a duration of 24 camera frames. This is equivalent to the time taken for the robot to take three half-steps under normal circumstances.

To begin, the robot reaches forward and slightly inwards with the front legs while squatting back with the rear legs. This drops the robot body close to flat to the ground, and moves the robot back a little. As with the old forward kick, the initial step of reaching forward with the front legs may seem odd, but if the ball is trapped against the robots chest to begin, this has the effect of catching the ball with the front legs, setting it up for the kick. This step lasts for 12 camera frames, giving the robot plenty of time to lean back and collect balance, ready to push forward with the kick.

Once it has set up for the kick with the first step, the robot then raises its front legs, ready to drop them on the ball. Unlike the old forward kick, the head is not used to hold the ball down in this step. This is because the timing has

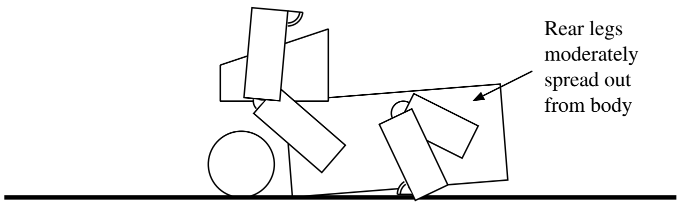


Figure 5.36: The second step of the aimable forward kick.

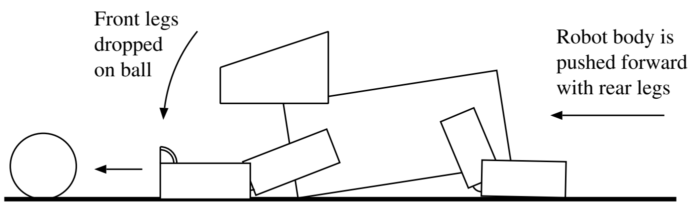


Figure 5.37: The final step of the aimable forward kick.

been changed for this and the previous step, removing the need to use the head. Whereas this step took 12 camera frames in the old forward kick, it only takes 6 camera frames in this version.

Finally, the robot drops the front legs on the ball while pushing forward with the rear legs. This last step is essentially unmodified from the old forward kick, apart from the movement of the rear legs. This step takes 6 camera frames, though increasing the duration of this step was tried. The result was a longer follow-through action on the kick, which did not produce any noticeable advantages.

Summary

The aimable forward kick has exactly the same duration as the old forward kick, but it is also more powerful and more flexible.

The main disadvantage of the aimable forward kick is instability. While the kick on its own is quite stable, it was found that when contested by other robots, the kicking robot could often be pushed over during the kick. This is because whereas the old forward kick placed the robot flat on the ground, the aimable forward kick tries to put some balance on the rear legs to push it forward. In other words, by trying to push forward with the front legs, the robot gives up some stability in exchange, making it more suspect to opposition robots.

The instability of this kick makes it particularly unsuitable for the goalie to use. If the goalie falls over, it is often very easy to score a goal before the goalie can recover to defend. Since the goalie is usually contesting the ball with opposition forwards, this can happen frequently if this kick is used by the goalie. Furthermore, the initial step of leaning the robot back before kicking the ball is particularly dangerous for the goalie. If the goalie traps the ball near the goal

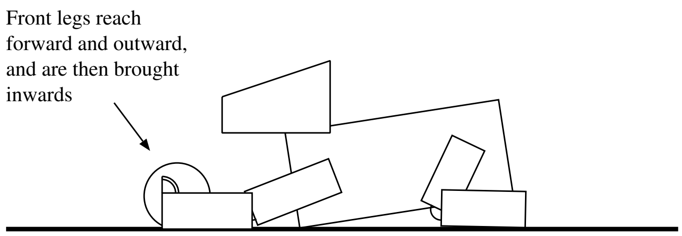


Figure 5.38: The integrated ball hold.

line and then executes this kick, it can sometimes bring the ball behind the goal line while leaning back in the initial step, causing an own goal.

In summary, the primary use of this kick should be for accurate shooting when the robot has some distance to opposition robots. This does limit the use of the kick significantly, but in general, almost all kicking actions are not effective when the ball is surrounded by other robots in any case.

5.3.6 Integrated Ball Hold

Background

The integrated ball hold was an experiment that could be applied to all kicks, but was only tried with the aimable forward kick. The initial step of the robot reaching forward with its front legs seem somewhat unreliable at times if the robot cannot consistently trap the ball. Integrating a ball gathering motion to the start of the kicking action was aimed at improving this.

Description

This action adds 12 camera frames to the duration of the kick. When added to the aimable forward kick, it extended its duration to 36 camera frames, or equivalent to the time taken for the robot to take four and a half half-steps under normal circumstances.

To begin, the robot leans forward by kneeling on its rear legs, while reaching forward and spreading its front legs. The spread of its front legs is at 45 degrees either way. This initial step takes 6 camera frames.

The robot then stays in this position, while bringing the front legs inwards. The front legs stay at full stretch during this process, producing a gathering motion that reliably collects the ball. This step takes another 6 camera frames.

With the gathering motion complete, the robot then executes the aimable forward kick, as described above.

Summary

The integrated ball holding motion is very reliable, allowing the robot to gather the ball without having it trapped against its chest. This allows for a much greater margin of error before executing a kick.

However, the integrated ball hold was never used in match play as it was far too slow to be useful. In match play, the robot often does not get much time to kick the ball before getting crowded by opposition robots, and so this method of trapping the ball to set up the kick proved unsuitable. Further attempts were made at speeding up the ball gathering motion, but in general, trapping the ball against the robots chest by charging up to it was preferred for its speed.

One possible application of this action that was not explored is as a kick-off skill. Since the robot kicking off has enough time to gather the ball to kick without worrying about opposition robots, this could have been used as a fast, reliable way to kick the ball deep into the opposition half at the start of play. However, kick off strategies were not explored this year, primarily due to time constraints.

5.3.7 Sideways Swipe

Background

The sideways swipe was devised as means of short range passing. It was designed to allow the robot to make short passes to teammate robots beside it, thus moving the ball around opposition robots before they can recover to defend. The turn kick can also produce a similar effect, but it is generally too powerful for short ranges.

Description

The sideways swipe consists of three steps, and has a duration of 27 camera frames. This duration is equivalent to a little below the time taken for the robot to take three and a half half-steps under normal circumstances.

To begin, the robot leans in forward while reaching forward and a little inward with its front legs. This is similar to the initial step of the integrated ball gathering motion, but without spreading the front legs. This traps the ball with the front legs, setting it up to be hit. This step takes 9 camera frames.

The robot then brings back one front leg, to prepare to hit the ball, while moving the other front leg to the side. This ensures that when the ball is hit, the other leg will not deflect the ball back. The rear legs are brought forward into a squatting position, and spread out from the body. This step takes a further 9 camera frames.

Finally, the robot swings its body while bringing in one front leg to hit the ball, while keeping the other away to the side. The rear legs actually swing the body a little, by pushing back with one leg while the other stays still. The swinging movement itself is small, but sufficient to allow consistent hitting of the ball. The duration of the recovery from this step seems lengthy, but it was found that if the recovery is shortened, the robot will often resume walking and deflect the tapped ball too early.

Summary

In general, the sideways swipe was not useful at all. Its major downfall was its instability, which seemed worse than the aimable forward kick. Like the aimable forward kick, the side swipe necessarily loses a bit of balance when swinging the robot body, and in the process, it becomes easier to push over.

At the competition, there were several implementations that resembled this skill. However, they seemed to require precise setup, as the robot simply takes a large swing with one front leg. This then requires having the ball a short distance in front of the robot, which is difficult to set up quickly and consistently.

5.3.8 Strong Forward Kick (aka "Chunky Kick")

Background

Casually called the "chunky kick" by the team, this is a reliable and powerful kick. Based on the old forward kick, this was a second attempt at utilising the rear legs during the kicking motion to produce more power.

A second version of this kick was also used in the competition, which was only modified in that the dropping angle of the front knees was toned back slightly for more reliability. This was originally done for the goalie, since the goalie specifically needed the kick to be as close to 100also preferred by the forwards.

Description

The strong forward kick is composed of three steps, and is very similar to the old forward kick. It has a duration of 24 camera frames, equivalent to the duration of three half-steps under normal circumstances. It is only slightly different to the old forward kick.

To begin, the robot drops to the ground while reaching forward and slightly inward with the front legs, while spreading its rear legs, with the knees bent. This is identical to the start of the old forward kick, except that in this case, the duration of the initial step was modified to 12 camera frames instead of 6.

The second step is also very similar to that of the old forward kick. The robot raises its front legs over the ball, ready to hit it down. The differences between this step and its counterpart in the old forward kick is that the head is not used to hold the ball down, and the duration has also been changed, from 12

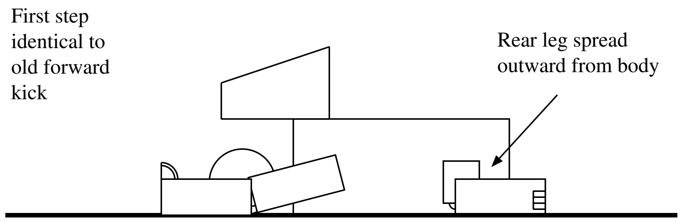


Figure 5.39: The initial step of the strong forward kick.

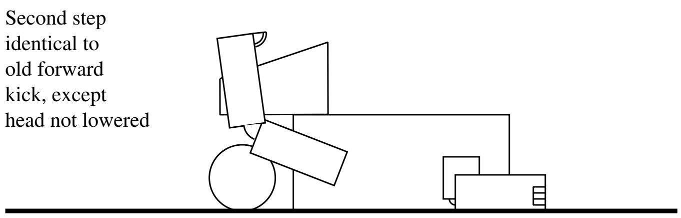


Figure 5.40: The second step of the strong forward kick.

camera frames to 6. Generally, this was because the kick was found to be reliable without using the head under the altered timings for the first two steps.

Finally, the robot drops the front legs on the ball, while bringing the rear legs inwards with the knees bent. This has the effect of raising the back of the robot, as the robot moves from lying flat on the ground to kneeling with the rear legs. Thus, the angle at which the front legs are brought down is also compensated for the slant of the robot body.

Despite the minimal changes that were made to the old forward kick in developing this one, this kick is considerably more powerful, even more powerful than the aimable forward kick. In addition, the robot is very stable for the duration of the kick, making it safe to use in situations where the robot may be in some risk of being contested by opposition robots while executing the kick.

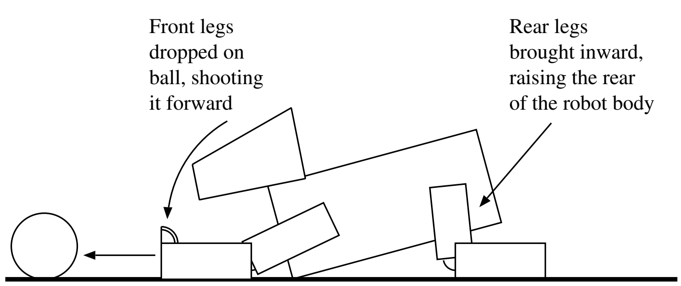


Figure 5.41: The final step of the strong forward kick.

Summary

The power and stability of this kick made it the preferred choice in most situations for long range kicking. Its stability makes it good for the goalie, since the goalie does not need to worry about being pushed over by opposition robots when trying to clear the ball with this kick.

In general, it was surprising that the small changes made to the old forward kick made such a difference in effectiveness.

5.3.9 Fast Kick (aka "Lightning Kick")

Background

The fast kick was devised late in development, originally intended to fill the range gap between the chest push and the strong forward kick. The chest push executes quickly but has short range, while the strong forward kick is relatively slower but has very long range. The fast kick was intended to be a middle range kick with faster execution than the strong forward kick. The end result, however, was a fairly powerful kick that was slightly faster than the chest push, but requiring a more restricted setup. Its surprisingly fast execution time earned it the nickname "lightning kick" among the team.

Description

The fast kick consists of just two steps, and has a duration of 16 camera frames, 2 fewer than the chest push. This is the same duration as that for the robot to take two half-steps under normal circumstances.

Before looking at the locomotion level actions, the setup for this kick should be discussed first. The setup is more specific than the other forward kicks, in that the fast kick does not execute correctly if the ball is held first. The fast kick saves time over the other forward kicks by not having an initial ball holding step, but this requires that the robot traps the ball with only the first part of the standard ball trapping sequence; that is, the robot must trap the ball under the head by approaching it while ball tracking, and then, without bringing the front legs around the ball, it must immediately go into the fast kick execution.

To begin, the first step has the robot look almost straight down, while raising the front legs and spreading the rear legs, with the knees bent. If the kick is set up correctly as described above, the robot will already be looking down on the ball when this first step is executed, and thus the head will simply stay down, trapping the ball while the front legs are raised. Another way to look at it is to consider the ball trapping trigger as the first step; that is, the robot begins the kick by trapping the ball under the head in a normal walking stance. In any case, this step of raising the front legs to prepare to kick takes 10 camera frames.

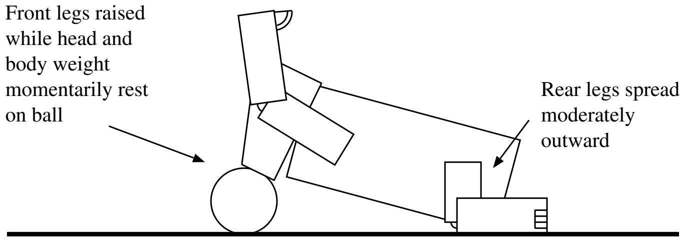


Figure 5.42: The first step of the fast kick.

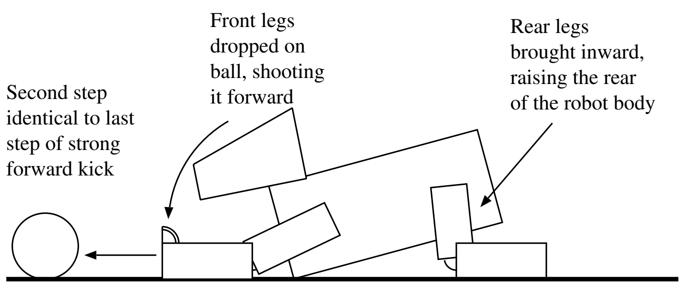


Figure 5.43: The last step of the fast kick.

The second step then merely brings the front legs down on the ball, shooting it forward. The rear legs are now brought in while keeping the knees bent, the same as the strong forward kick. The head is raised in this step, since it must not trap the ball when it is hit.

Summary

The main advantage of the fast kick is speed. Its minimal execution time gives opposition robots less time to crowd the ball when setting up the kick, and it also covers good range, quickly moving the ball up the field.

The main disadvantage of the fast kick is that the setup needs to be more precise than the other kicks. Because it does not execute reliably if the robot holds the ball with the front legs before executing the kick, it restricts the use of the kick to situations where the robot and the ball are already lined up with the target. It does not work if, for example, the robot tries to hold the ball, localise, and then kick, or if the robot tries to turn with the ball first to line up a target. In these cases, if the fast kick is executed, the front legs will often drop down on the ball but instead of shooting the ball out, they trap the ball instead, rendering it ineffective.

While the disadvantages restrict the potential uses of the kick, it is excellent for quickly moving the ball up the field. In such cases, the robot does not need to line up a particular target, and instead just aims to kick it in a general direction. The fast kick is ideal for these situations, and the strategy can then use the other kicks once it needs to aim for the goal.

5.3.10 Other Ideas

There were several ideas that were considered but never implemented. Generally, these were inspired by watching competition videos from the previous year, where many teams showcased a variety of inventive kicking skills.

Sideways head swipe

The sideways head swipe was used by several teams in the competition both this year and last year. To execute, the robot looks down and away to one side, and then swings the head around to the other side.

In general, this appears to be a useful skill to possess. However, the obvious problem with this skill is the setup. In order to execute this reliably, the ball needs to be in front of the robot, but a small distance away. Since the robot never traps the ball at any point during the execution of this skill, it has to rely on the ball staying where it is while it swings the head, something that becomes less likely with the presence of opposition robots.

Furthermore, since the turn kick can be used for the same purposes with greater power and easier setup, the sideways head swipe was not pursued further.

Varying forward kick angles

This idea came up very late in development, and was never implemented. The idea is to vary the angle at which the front legs come down in the forward kicks, such that the legs come down at different angles. This would theoretically result in the ball being kicked forward at slight angles. For example, if the left leg was brought down almost straight, and the right leg was brought down with the knee fully bent, the ball would shoot towards the right.

This concept remains open to investigation, but it may provide a useful alternative means of aiming the kick to some extent.

Dive kicks

Several teams have exhibited this skill, which has the robot dive forward on the ball. Most implementations simply have the robot fall forward on the ball, shooting it with excellent power but almost no accuracy. The Carnegie Mellon team integrated a ball holding action to the start of their version of this kick, making it more reliable, but very slow. However, the power of this kick is excellent, since the robots body weight is used, and can push back defending robots to a small degree.

5.4 Locomotion Parameters

In order to actually utilise the walking routines, a set of parameters is required to specify the stance and movement dimensions of the walk. These parameters are passed through to the walking routines, and are then used in subsequent calculations.

5.4.1 Parameter summary

A summary of the parameters follows.

- `walktype (int)` - specifies the actual type of locomotion action to execute. Walktypes are basically definitions, used to decide whether to execute a particular walking style or kicking action. With only the exception of the aimable forward kick, the kicking actions ignore all other parameter values.
- `forward (double)` - specifies the distance in cm that the robot should move forward in each half-step. Negative values move the robot backwards.
- `left (double)` - specifies the distance in cm that the robot should move to the left in each half-step. Negative values move the robot to the right.
- `turnCCW (double)` - specifies the angle in degrees that the robot should turn counter clockwise per half-step. Negative values turn the robot clockwise.
- `PG (int)` - specifies the time taken to complete a half-step, in units dependent on the variable `numOfFrames`, defined in `PWalk.cc`. `numOfFrames` is the number of walk locus end-points calculated per leg for each camera frame, with

  PG = `numOfFrames` * number of camera frames per half-step.

  PG must be divisible by `numOfFrames`. "PG" is an acronym for "period ground", meaning the time taken for the robot paw to move along the ground, or the bottom edge of the walk locus.
- `hf (double)` - the vertical height in mm from the shoulder joint to the ground. "hf" is an acronym for "height forward". The point on the ground vertically below the shoulder joint is called "shoulder ground".
- `hb (double)` - the vertical height in mm from the hip joint to the ground. "hb" is an acronym for "height back". The point on the ground vertically below the hip joint is called "hip ground".
- `hdf (double)` - the height in mm that the robot lifts its front paws when walking, that is, the height of the walk locus for the front paws. "hdf" is an acronym for "height delta forward".

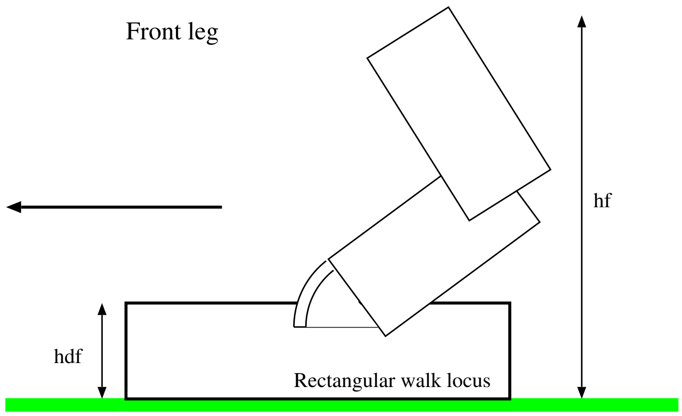


Figure 5.44: A visual representation of hdf and hf.

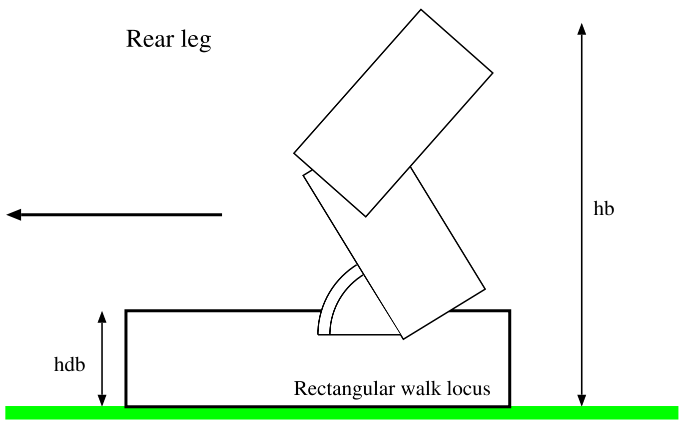


Figure 5.45: A visual representation of hdb and hb.

- `hdb (double)` - the height in mm that the robot lifts its back paws when walking, that is, the height of the walk locus for the back paws. "hdb" is an acronym for "height delta back".
- `ffo (double)` - the home position of the front paws in mm forward from shoulder ground. "ffo" is an acronym for "front forward offset".
- `fso (double)` - the home position of the front paws in mm sideways from shoulder ground. Larger values spread the legs outwards, smaller values bring them in closer. "fso" is an acronym for "front sideways offset".
- `bfo (double)` - the home position of the back paws in mm forward from hip ground. "bfo" is an acronym for "back forward offset".
- `bso (double)` - the home position of the back paws in mm sideways from hip ground. Larger values spread the legs outwards, smaller values bring them in closer. "bso" is an acronym for "back sideways offset".

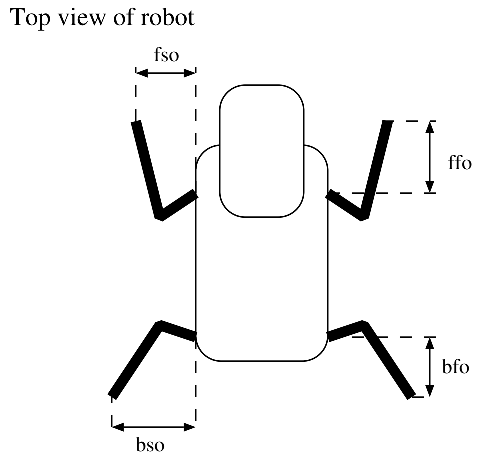


Figure 5.46: A visual representation of ffo, fso, bfo, and bso.

In addition to these, there are parameters for controlling the head movement:

- `headtype (int)` - the type of head movement, can be either none, relative, or absolute.
- `tilt (double)` - the vertical angle of the head. Negative values make the robot look down, positive values make the robot look up. Note that this is necessarily relative to the robot body slant.
- `pan (double)` - the horizontal angle of the head. Negative values make the robot look to the right, positive values make the robot look to the left.

There is no parameter for controlling the roll of the head, nor has there been a need for one.

Control of the head is separate from control of the legs. While new walk parameters can be fed to the legs only at intervals of PG time, the head can accept parameters on every frame. This allows for responsive visual tracking.

5.4.2 Exploring the walk parameter space

With 12 variables affecting the walk stance and movement dimensions, there is vast exploration that can be conducted to try to improve on the existing calibrated walk.

Explorations of the parameter space can produce stance variations, as described above in the stance variation section. However, the parameters also control the stepping variables, that is, how frequently steps are taken, and the dimensions of the walk locus.

An experiment with the stepping variables was to further increase the stance height, and to try to take fast, small steps. This did not produce a useful walk, but it was interesting nonetheless to observe a possibility of exploring the parameter space.

Joint angle limits

Exploration of the parameter space is sometimes limited by the joint angle limits. Specifically, the parameters supplied to the walking routines may result in some points of the walk locus being physically unreachable. This can result in impossible leg joint values, and this causes a safety mechanism to tuck the legs of the robot against the body.

It is possible to examine whether a given parameter set causes this to occur by greatly increasing the value of PG (for most purposes, a value of around 200 is enough). This makes the robot move through the walk locus very slowly, and thus an unreachable point on the walk locus with a small PG becomes a group of unreachable points with the larger PG. This then means the safety mechanism will be activated for a longer period, thus showing the offending point(s) on the walk locus.

One thing that must be realised about this is that the activation of the safety mechanism is not necessarily detrimental to a walking action. With realistic values for PG, unreachable points on the locus may cause the robot to try to tuck its legs inwards for a brief split second, making a barely noticeable difference. Additionally, even if this effect is noticeable, it may not necessarily slow down the walk. In short, it sometimes appears that improving the effectiveness of the walk is counter-intuitive, and effects such as this may actually improve the walk. It is likely that this is due to the particular details of the robot (e.g. the spring loaded claws, the weight of the robot being biased towards one side due to battery weight, etc.).

5.5 Future Improvements

The locomotion module has been steadily improved since its original implementation in 2000. However, there is still considerable room for improvement, with some ideas suggested below.

5.5.1 Walk styles

The walking style seems to have reached a local maxima, with the existing stance and rectangular locus being used since the original implementation of the locomotion module.

The rectangular locus was thought to be optimal until the development of the trapezoidal locus. Likewise, it is also possible that the trapezoidal locus is also not the optimal locus shape. A possibility here would be to apply machine learning methods to dynamically learn optimal locus shapes for walking under different circumstances (i.e. different combinations of forward, sideways, and turning parameters) and with different goals (e.g. speed, stability, steadiness of camera), with the end result likely to be a trade-off between the goals.

In addition, the walking parameters such as the height of the robot body may also not necessarily be optimal for different circumstances. This may also be investigated further.

5.5.2 New kicking skills

Although much experimentation was done on kicking skills this year, this area is always open to new ideas. For example, the idea of varying the rate at which the front legs are dropped in the forward kicks was only suggested late in development, and has not been tried yet.

5.5.3 Get-up routine

The present implementation still makes use of the default Sony get-up routine when the robot falls over. This default routine is somewhat slower than it needs to be, and thus it would be useful to develop a customised get-up routine.

5.6 Review of Related Work

There is a good amount of related literature in the fields of quadruped robots and legged locomotion, reflecting the interest in legged robots. In particular, there is a considerable amount of research now in humanoid robot walking. While it bears some relation to this field, humanoid walking is made considerably more difficult since keeping a humanoid robot balanced is much more difficult than in the case of a quadruped.

The following is a discussion of some other research in quadruped walking.

5.6.1 Evolutionary Algorithms

Given a parameterised approach in which walking routines are driven by an input parameter set, applying machine learning methods to attempt to find an optimal parameter set is a natural approach.

Hornby, et. al in [HFT+99] and [HTY+00] suggest applying evolutionary algorithms on a set of parameters to autonomously learn gaits, with different purposes in mind.

In [HFT+99], this approach is applied in order to evolve a pace and trot gaits, with speed and accuracy as the main objectives. That is, in the evolutionary algorithm, individuals of the evolving population that were slow or tended to deviate from walking in a straight line were penalised. The result was that efficient pace and trot gaits were evolved, outperforming sample pace and trot gaits implemented by hand prior to the experiment.

In [HTY+00], a similar approach is applied in order to evolve more robust gaits. The authors observe that the gaits developed in [HFT+99] were not entirely robust, in that they tended to be ineffective when tested on different robots or on different surfaces. A similar test environment was employed, except that the placement of artificial ridges in the surfaces was tried. Consequently, successful individuals of the evolving population raised the legs higher than normal, in order to avoid getting caught or tripping on the ridges. In parallel, the same test environment without the ridges was tried. The result was that the gaits evolved on the surface with ridges turned out to be more effective on other surfaces.

5.6.2 Omnidirectional Locomotion

Hengst, et. al [HIPS01] discuss the principals and development of the locomotion module used by the UNSW team in RoboCup 2000. As described earlier, much of the underlying theory of the walking routines have not changed since its original implementation, outlined in this paper.

While this research was particularly focused on the problem of locomotion for the RoboCup legged league, it discusses a model of robot locomotion that is more maneuvrable than others. Other research in the field tends to focus on the problem of developing an efficient gait for walking forward or turning, but often does not address the issue of omnidirectional movement.

5.6.3 Bounding and Pronking Gaits

Yamamoto, et. al [YFdL+01] discuss development of bounding and pronking gaits using a modified version of the Sony AIBO robot used in the RoboCup legged league. The robot is equipped with spring loaded legs in this experiment, allowing it to move towards a more natural running motion rather than the walking discussed previously.

As a reminder, the bounding gait synchronises movement fo the front legs together and the back legs together. The pronking gait, not discussed up to this point, moves all four legs in synchronisation. This is obviously only viable if the robot can actually lift off the ground.

5.6.4 Other Models

Huber and Grupen [HG98] discuss combining reinforcement learning with a Discrete Event Dynamic System, an abstract model of behaviour, in order to learn gaits. In summary, the exploration of the reinforcement learning is supplemented with additional domain knowledge from the abstract model. The model is used here to prevent the robot from falling over, thus allowing it to learn autonmously without human supervision. This was a problem also faced by Hornby, et. al in [HFT+99] and [HTY+00], but was solved in those cases by giving a fitness score of 0 to any individuals of the evolutionary population that caused the robot to fall over. The robot would detect when it had fallen over using accelerometers, and then use its own routines to get up and resume evaluation of the remaining population.

Lewis [Lew02] discusses the use of Central Pattern Generators and Adaptive Modules, with the goal of simply learning stable gaits. The analogy drawn in this study is that a foal learns to walk and run within minutes of being born. By trying to model this learning behaviour, the author reports that the system learns to walk within several minutes.

Chapter 6

Skills

Chapter authors: James Brooks (improved turn kick section), Min Sub Kim (other sections).

6.1 Overview

In addition to the basic skills handled by the locomotion module, there is an additional set of skills that are covered by the strategy module. Generally, these skills are more complicated than the skills provided by the locomotion module, as they are designed to interact more with the game. This is actually the main reason for these skills not being handled by the locomotion infrastructure; they usually require access to game world objects in order to interact with them. On the other hand, skills in the locomotion infrastructure simply move the robot without any regard for the game environment.

6.2 Ball tracking

6.2.1 Background

One of the most fundamental skills of the game, ball tracking keeps the robot focused on the ball. It was borrowed from last years development and was kept largely unchanged.

6.2.2 Description

The ball tracking routine is actually executed via a sequence of 3-dimensional transforms. These transforms output the values for the tilt and pan of the robots head.

To begin, the position of the ball in 3d space relative to the camera is found. In this coordinate system, the x-axis is horizontal in the camera, the y-axis is vertical in the camera, and the z-axis is the depth. Thus, the values of x, y, and z are set as follows:

$$\begin{aligned} x &= distance \times \tan heading \\ y &= distance \times \tan elevation \\ z &= distance \end{aligned}$$

Next, the values of y and z are adjusted, by translating them according to the length of the face and the neck.

$$\begin{aligned} y &\leftarrow y + necklength \\ z &\leftarrow z + facelength \end{aligned}$$

The position is then rotated around the y-axis according to the current head pan. The rotation matrix for this is:

$$\begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

thus producing

$$\begin{aligned} x &\leftarrow x \times \cos pan - z \times \sin pan \\ y &\leftarrow y \\ z &\leftarrow x \times \sin pan + z \times \cos pan \end{aligned}$$

Then the position is rotated around the x-axis according to the current head tilt. The rotation matrix for this is:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

thus producing

$$
\begin{aligned}
x &\leftarrow x \\
y &\leftarrow z \times \sin tilt + y \times \cos tilt \\
z &\leftarrow z \times \cos tilt - y \times \sin tilt
\end{aligned}
$$

The position is then clipped, to account for the head joint limits. This translates to setting z to the neck length if it is smaller.

The required head then found by finding the value required in the yz plane to align the ball position. It is also compensated for the neck length:

$$tilt = \arctan \frac{y}{z} - \arcsin \frac{necklength}{\sqrt{z^2 + y^2}}$$

Finally, the yz plane is rotated to align the ball position on the xz plane. The required pan component can then be found:

$$
\begin{aligned}
z &\leftarrow z \times \cos(tilt - y \times \sin(tilt)) \\
pan &= \arctan \frac{x}{z}
\end{aligned}
$$

In the actual implementation, some values of pan and tilt are taken as negative. This is because the x-axis in this coordinate system is positive towards the left of the camera, whereas conventionally it is positive towards the right.

There are actually two versions of ball tracking commonly used - one that tracks the middle of the ball, and one that tracks the top of the ball. The difference in implementation is simply the calculation of the first y value in the above transforms, by changing the value of the elevation.

Tracking the middle of the ball has the advantage of being less likely to lose the ball. To put simply, since the ball is kept in the middle of the robots view, it is less likely that the ball will travel far enough between camera frames to make the robot lose sight of it. This is most noticeable in the case where the ball is moving towards the robot.

Tracking the top of the ball has the advantage of generally seeing more landmarks. Because the robot tends to look higher than if it were tracking the middle of the ball, it is more likely to catch glimpses of landmarks. This was the version preferred in strategy, primarily to reduce the frequency of lost localisation.

6.3 Trapping the ball

6.3.1 Background

Trapping the ball can be considered to be the most fundamental prerequisite to consistently executing kicking skills. In short, the basic setup for virtually all the kicking skills is to trap the ball directly in front of the robot, with the ball between the robot's front legs, up against the robot's chest. Consequently, being able to perform this skill reliably and consistently is critical to the overall strategy.

In addition to reliability, for this skill to be useful in strategy, there is the secondary requirement of speed. The main reason for this is that in match play, if the robot takes too long to gain control of the ball, it gives too much time for the opposition robots to surround the ball. This effectively renders the kicking skills useless, as it is virtually impossible to kick the ball out when surrounded. Another reason for needing this skill to be fast is that there is a ball holding time limit applied in match play, allowing forwards to hold the ball for 3 seconds, and the goalie to hold the ball for 5 seconds. Since the ball holding time limit starts from when the robot's front paws are around the ball, the robot has little time left after trapping the ball to decide and execute a kick. It follows then that trapping the ball must be both reliable and efficient.

6.3.2 Description

This skill can be considered as a series of steps, although not all the steps need to be completed to successfully execute a kick, depending on the kick that is selected. These steps are:

- Approach the ball at a fast walk speed, while tracking the ball. As the robot moves closer to the ball, the head tilt of the robot decreases until it is looking down on the ball. At this point, the ball is trapped under the robot's head.
- With the robot still looking down on the ball, the front legs are brought around the ball. At this point, the ball is very tightly held.
- With the ball firmly held between the front legs, the robot can lift its head to localise if necessary.

In general, for an efficient setup, only the first step of the sequence is required. The second and third steps are only really required when the robot is lost and needs to relocalise, or to use ball handling skills.

The critical parameter in this skill is the trigger value for the head tilt, used to decide when the robot has trapped the ball close enough to execute a kick. This parameter requires a fair degree of tuning, and is further complicated by different ball tracking skills and different walking styles. Essentially, there were

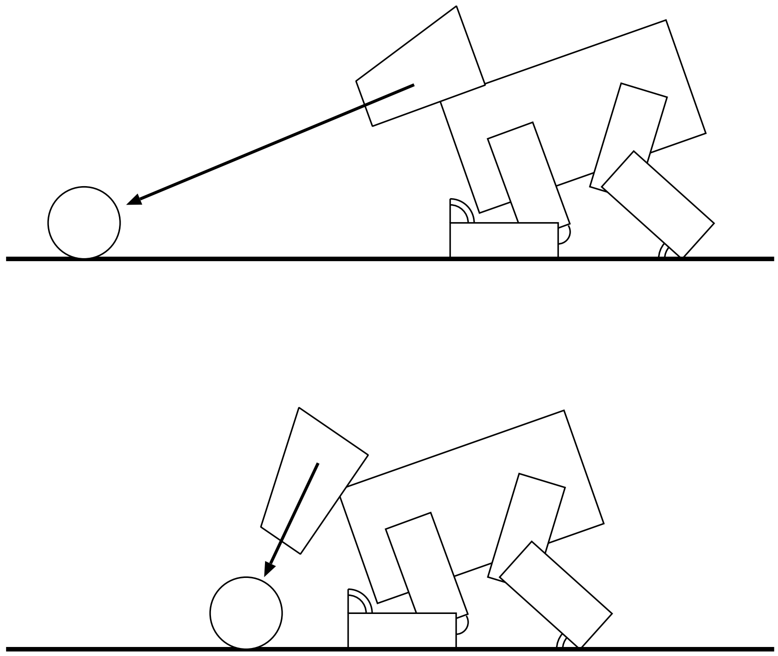


Figure 6.1: As the robot moves closer to the ball while tracking it, its head moves to a lower angle to look down on the ball. Once this angle is steep enough, the robot decides that it is close enough to the ball to trap it.

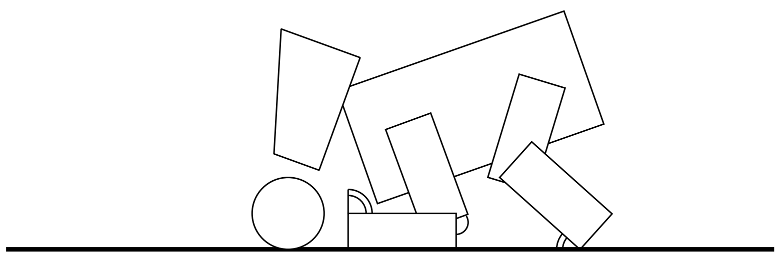


Figure 6.2: At this point, the head tilt of the robot is steep, looking almost straight down on the ball. This is the trigger used to decide if the robot is close enough to the ball to trap it. From this point, the robot is able to execute most kicking skills.

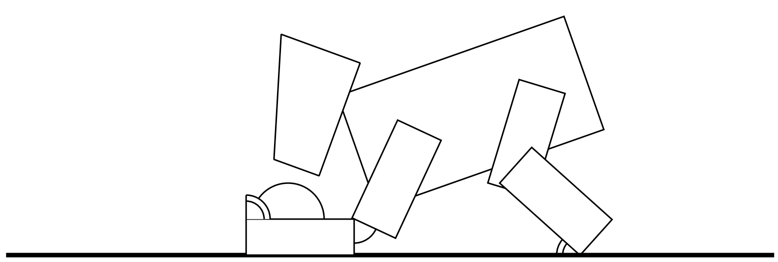


Figure 6.3: With the ball tucked under the head of the robot, it can bring its front legs around to fully trap the ball. This holds the ball very tightly.

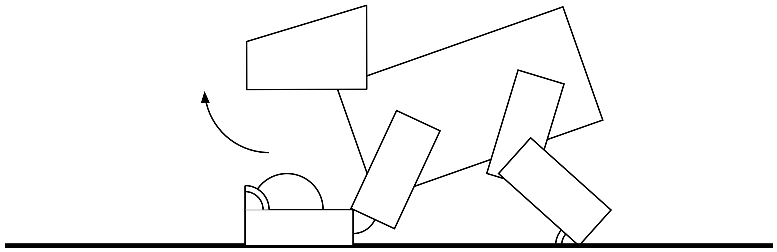


Figure 6.4: With the ball held by the front legs, the robot can safely raise its head to look around. This allows it to localise or visually aim for a target.

four combinations of ball tracking and walking styles that were considered in tuning this parameter:

- Tracking middle of the ball, regular walk (rectangular locus): this is the most reliable combination. The combination of the robot focusing on the middle of the ball and the steady walk make this setup very reliable. The trigger tilt is set to a moderately steep angle.

- Tracking middle of the ball, trapezoidal walk (trapezoidal locus): this is a mostly reliable combination. The trapezoidal walk is less steady than the regular walk, and shakes the camera more, making the ball tracking less reliable. The trigger tilt needs to be set to a steeper angle than with the previous combination.

- Tracking top of the ball, regular walk: this is a mostly reliable combination. Tracking the top of the ball has the disadvantage of occasionally losing sight of the ball right underneath the robots vision, particularly if the robot is walking fast towards the ball, or if the ball is moving towards the robot. There are however strategical reasons for tracking the top of the ball, to be discussed in the ball tracking description. The trigger tilt is set at a higher angle than the previous combinations.

- Tracking top of the ball, trapezoidal walk: this is the worst combination. The robot can easily lose sight of the ball with this combination, making it effectively useless.

Tuning the trigger tilt is basically a trade-off between speed and reliability. An angle that is too steep is very reliable, but often causes the robot to walk into the ball longer, wasting time. An angle that is too high will result in the trigger being reached earlier, but the ball may not be close enough to the robot to be reliably trapped.

6.4 Turning with the ball

6.4.1 Background

Once the ball is trapped, it is sometimes useful to be able to turn with the ball while keeping it held. For example, the robot may need to turn to face the target goal to aim.

There are two versions of this skill, one that is slow and one that is fast. In general, the slower version is rarely useful in match play since the robot often needs to act quickly with the ball. However, it does have certain advantages over the faster version. Both versions, along with their advantages and disadvantages, are described below.

6.4.2 Description of slower version

In the slower version, the robot fully traps the ball, covering all three steps in the ball trapping sequence outlined above. That is, the robot firstly traps the ball under its head, and then brings its front legs around to grasp the ball, and then raises its head. At this point, the robot then rotates around the ball by taking sideways steps with its back legs, while keeping the ball held with its front legs. While turning with the ball in this manner, the robot is able to stop and immediately execute most kicking skills reliably.

This version of the skill was used by the UNSW team last year in one of the challenges. That version was used as the starting point for this skill this year, but was modified to be faster. In the challenge, the focus had been on reliability, as speed was not an issue (the normal ball holding rules do not apply in the challenge competition). When turning with the ball, the robot took slow, large sideways steps with the back legs, steadily turning to aim. This was modified by reducing the step size by a small proportion, and increasing the stepping rate by a large proportion, to the point where it matches the stepping rate of the regular walk. The robot is thus able to overall turn faster, by taking more smaller steps. Furthermore, this change did not compromise reliability, making it better overall.

Despite the above changes to improve speed, this version is still too slow to be useful in match play most of the time. The robot still needs to take too much time to trap the ball, often giving opposition robots enough time to crowd the ball. Overall, the time required by this version of the skill is too often more than the robot can afford.

The primary advantage of this version is that the robot turns with its head raised. This way, the robot is better localised while turning and can aim more reliably by visually identifying the target.

This version of the skill was used in the ball collection challenge this year, outlined in that section of the report. There were no ball holding rules in the challenge, no opposition robots to deal with, though the robot needed to get a good aim on the goal, making it an ideal situation to use this skill.

6.4.3 Description of faster version

In the faster version, the robot skips the latter steps of the ball trapping sequence, that is, it does not actually hold the ball with its front legs. Instead, the robot simply traps the ball with the head by approaching it and tracking it, and then proceeds to turn around the ball while keeping the head down. While turning, the front legs are brought forward further than normal, thus keeping the ball tucked under the head and between the front legs. Odometry is relied on to decide when the robot has turned the desired angle.

The main advantage of this skill is speed. The robot moves directly from trapping the ball to turning with very minimal delay. In addition, the ball stays trapped in a good position to execute most kicks, allowing the robot to go directly into a kick when ready. Two extensions of this skill are the spin kick and spin chest push, which are simply executed by going into either a strong forward kick or a chest push after turning.

The main disadvantage of this skill is that the robot needs to rely on odometry. Thus, if the robot is blocked by obstacles such as opposition robots while turning, the odometry will be incorrect and thus the robot may shoot in the wrong direction. Nevertheless, the speed of this version makes it more desirable in match situations.

6.5 Walking sideways with the ball

6.5.1 Background

A common problem that occurs during match play is having kicks blocked by opposition robots moving in front of the ball. Walking sideways with the ball is a skill that was originally designed to remedy this problem. The idea was that the robot would trap the ball, and then walk sideways while pushing the ball along with it, thus sidestepping an opposition robot.

6.5.2 Description

This skill is executed by simply trapping the ball and then walking sideways. To keep the ball in control while walking sideways, the robot's head can be kept down on the ball. Otherwise the robot's head may be kept up to decide if it has side-stepped the opposition robot far enough to kick, but this makes the skill less reliable, as the ball is more suspect to roll out.

In general, this skill did not prove useful in strategy for its intended purpose. A major problem was that the robot often had to deal with more than one opposition robot, and so sidestepping one of them was usually insufficient. However, this skill was used effectively in the ball collection challenge, where it was used to move the robot away from the side edges, to make room for the robot to turn and aim for the goal.

6.6 Goalie block

6.6.1 Background

The goalie block is a straightforward skill designed to block incoming kicks. It was modified slightly from last year's version.

6.6.2 Description

The goalie block is probably the simplest strategy level skill. The robot simply lowers its front body, such that its chest touches the ground. At the same time, the front legs are spread sideways at a wide angle, covering a wide span. The robot is free to move its head as well, allowing tracking of the ball as it comes in.

The skill is actually executed as a specialised walk. In short, the regular walking routines are called, with specific parameters that put the robot in the goalie block stance. Because it is executed in this way, the goalie block can be of variable duration, and the robot can quickly recover to start moving again afterwards.

There was also a second version of the goalie block that was used in one of last year's challenges. The second version was a fixed special action in the locomotion infrastructure. It was used in the goalie challenge, in which the goalie had to move into position in front of the goal, block an incoming ball, and then clear the ball. While suitable for that challenge, it was not suitable for match play due to its fixed duration and slow recovery, that made it far less flexible.

6.7 Getting behind the ball

6.7.1 Background

Getting behind the ball is a fundamental defensive skill, used to position the robot in a safe location behind the ball before trying to push it forward.

In general, the use of this skill in strategy is cut off up to getting beside the ball, at which point the robot traps the ball and executes a turn kick. This was

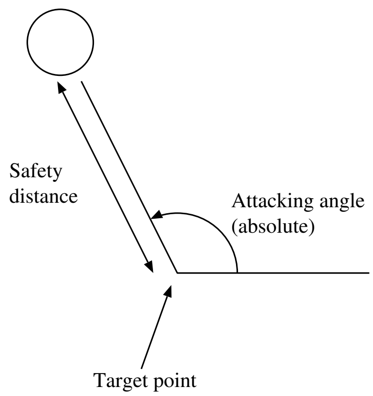


Figure 6.5: The calculation of the target point. The attacking angle is defined as an absolute angle. The safety distance is the distance to set the target point from the ball.

preferred over moving completely behind the ball as it was faster, but is more risky since the robot is more likely to accidently knock the ball backwards. This was the preference of both last year's and this year's strategy.

6.7.2 Description

The implementation of this skill owes its origins to the implementation in the 2000 team's strategy, and is heavily based on it.

This skill is parameterised with two values, the attacking angle and the safety distance. The attacking angle is the heading that the robot eventually wants to face the ball. For example, to move behind the ball and line up the goal, this angle should be equal to the heading of the goal from the ball. The safety distance is the minimum distance to the ball at which the robot should move around it. This value is tweaked according to how aggressively the robot should circle the ball, but it is best to set it at a reasonable value to safely move around the ball without accidentally hitting it back. With these two parameters, the target point can be calculated - this is the position that the robot should move to, according to the attacking angle and the safety distance.

With the target point calculated, the next part to be calculated is the next step to be taken in order to reach the target point. This could be done by simply moving the robot directly to the target point. However, this approach fails as this can easily produce a direct path that goes through the ball, thus failing to move the robot around it. In order to make the robot move around the ball, the next step is determined by calculating a circling point. The circling point is a point that is the same distance away from the ball as the target point, but based on the robot's heading to the ball, offset by an angle of 70 degrees. In the implementation, two circling points are calculated - one offset by +70 degrees from the target point, and the other offset by -70 degrees.

With the target point and circling points calculated, the next step to be taken

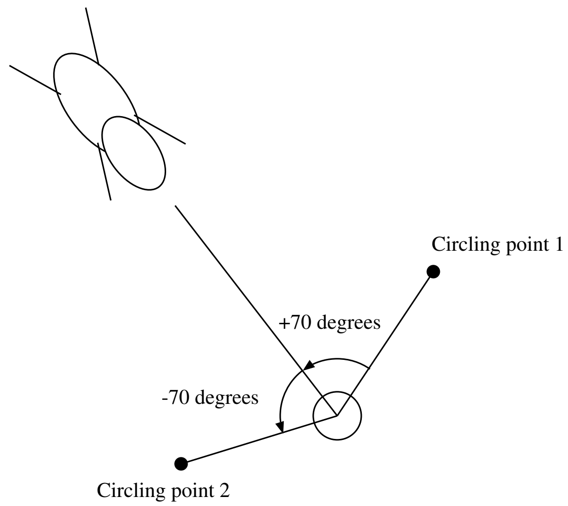


Figure 6.6: The calculation of the circling points. They are calculated as offsets of +70 degrees and -70 degrees from the heading of the robot to the ball.

can finally be calculated. Firstly, one of the circling points is eliminated, depending on the half of the field that the ball occupies. This is to ensure that the robot moves the "correct" way around the ball, that is, that the robot does not walk into the side edges of the field. The next step is then calculated by moving the robot towards either the target point or the remaining circling point, whichever is closer. The forward and sideways parameters in the walk are used to move the robot towards that point, and the turn parameter is used to keep the robot facing the ball.

The final result of these calculations is a smooth circling motion that guides the robot around the ball efficiently. Since the circling point is calculated as an offset from the robot's heading to the ball, the robot never actually reaches the circling point. However, once the circling point moves to be further than the target point, the robot smoothly switches to moving to the target point. Furthermore, since the points are constantly recalculated, the robot is never locked into a fixed sequence of steps. This means that it can deal with the ball moving away, as it will simply chase the ball further to reach the circling point.

In terms of strategical applications, as mentioned above, this skill is often not used to move fully around the ball, but is still used heavily to retrieve the ball. Ending the circling approach prematurely and going into the turn kick is particularly preferable and efficient when the ball is on the side edge, as the robot quickly moves the ball along the side edge with the turn kick.

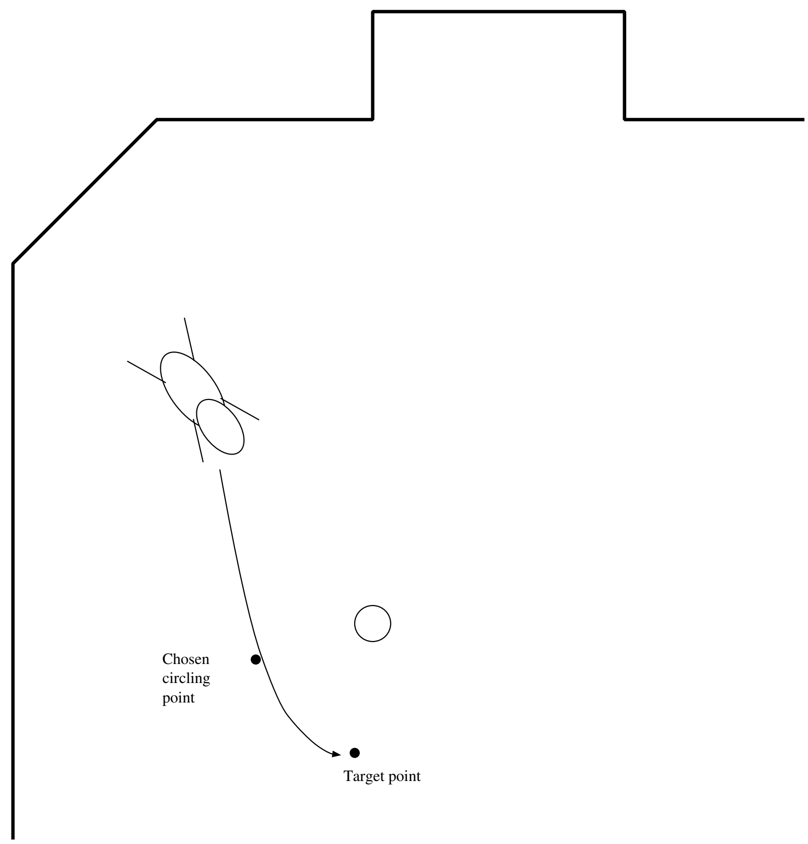


Figure 6.7: The previous diagram, with one circling point discarded. The approximate path taken by the robot is shown.

6.8 Paw Kick

6.8.1 Background

The paw kick is a reworking of the paw dribble skill from last year. In short, the paw kick works by simply walking the robot in the direction of the ball with a small offset to the side, so as to hit the ball with the robots paw. Since the paws face forward with the regular low stance, the ball is hit consistently forward, with deviations resulting only due to inconsistencies on the ball or field.

The paw kick was developed by Andres Olave, and a more detailed description can be found in his thesis document [Ola02].

6.8.2 Description

In addition to the paw kick, there is a second version called the lateral kick. The lateral kick is virtually identical to the paw kick, except that it hits the ball while walking sideways, thus hitting it with the side of the lower leg.

The paw kick is executed by lining up the ball in the desired direction with the correct offset and simply walking forward. The robot needs to commit to which paw it will hit the ball with, to prevent problems caused by indecision at the last moment.

Like the get behind ball skill, the movement parameters are constantly recalculated through execution. Thus, the robot is not locked into a fixed sequence of steps, allowing movement correction if, for example, the ball moves while the robot is approaching it.

The main advantages of the paw kick are its minimal setup time and speed. The robot kicks the ball without going through the ball trapping sequence, saving a lot of time. In addition, the paw kick is more appropriate than most other kicking skills when contesting the ball against opposition robots, since it is difficult to get a hold of the ball in that scenario. Lastly, the paw kick is able to get to the ball in cases where it may be less appropriate to hold it. For example, if the robot tries to trap the ball along the edges of the field, it needs to move around and approach it perpendicular to the edge. If the paw kick is used instead, it can efficiently hit the ball along the edge of the field without having to move around first.

The disadvantage of the paw kick is its inaccuracy over long ranges. However, since the use of the paw kick is primarily to push the ball quickly up the field, this disadvantage is not often prevalent.

6.9 Improved Turn Kick

6.9.1 Background

The *TURNKICK* is a movement that a dog can perform in order to kick the ball at a large angle (greater than 90 degrees) from the current facing direction. This type of kick is especially useful for a number of reasons. The kick execution time is relatively small and there is not much setup time or complexity in the setup manoeuvres. Another extremely useful attribute is that the ball can be kicked at quite large angles from our initial direction. This gives the dog the advantage of being able to approach the ball from almost any direction and then, without much manoeuvreing, move the ball at speed in the desired direction. This can be useful because opponent dogs may be in the vicinity and if we were to try to orient ourselves in the desired position so that we were lined up with the target before we grab the ball to kick then the ball is likely to be stolen by the opponent dog, or we are likely to simply bump or nudge the ball in the wrong direction.

The 2001 Robocup code contained the implementation of a turn kick. This turn kick was very useful in game situations however there were some observable problems with it. The turn kick itself was variably unsuccessful. Also, the way the turn kick was implemented, we had only very coarse control over the magnitude of the turn. Partly as a result of this coarse control, often the dog would turn and in the middle of the turning motion, the ball would come out prematurely from between the paws. The problem with this is that the dog would continue to turn and would not realise that the ball was actually not between the paws anymore, and as a result it would not be able to see the ball when the turn kick movement was finished. This is illustrated in Figure 6.8. This problem is a result of other factors aswell such as the dog not being able to see the ball when it is in the turn motion.

Because of these limitations with the old turn kick, we decided to start work on the development of a new turn kick. We wanted to have more control over where the kick would go, that is whether the kick would go at a very large angle (120+ degrees) or at a medium angle (90-120 degrees). We had a feeling that the current turn kick was sub-optimal and also the variability in it's success hindered it's use in gameplay.

6.9.2 Development

The TurnKick Parameters

It was decided that we would try bringing the front paws further forward during the kick so that the kick was less likely to fail from the aspect of premature ball ejection. This worked well. To do this we set the Pwalk ffo variable to 80.

We also attempted to execute a more powerful kick by increasing the speed at which the dog turned. Already the kick was executed at the normal 'limit' for

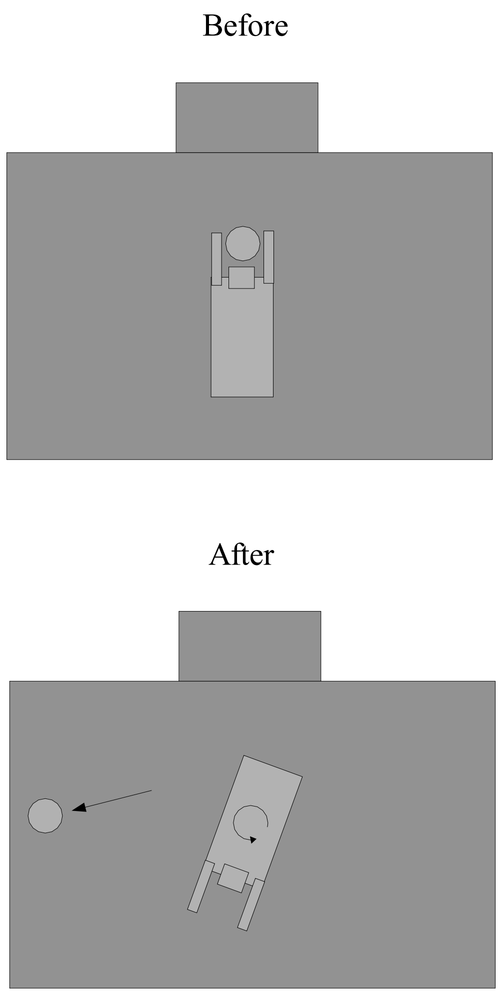


Figure 6.8: The dog has turned to the left and the ball has come out before the turn was completed.

the turnCCW walk parameter, but because of the fact that these are soft limits and are not enforced by the Pwalk module, we were able to drive the turnCCW varible larger to 50+ degrees. The results of testing different speeds of turning showed that 50 was the most powerful and most workable value as values above 50 produced leg movement positions which were too great for the leg motion to reach and thus provided close to erratic leg movement.

Further testing showed that some control over the direction of the kick could be maintained by varying the number of Pwalk cycles the turnkick was executed for. Two cycles of execution produced the medium turn kick, and three cycles produced the large turn kick.

There was also experimentation with executing the turn for only one cycle. This produced an interesting kick such that the dog could kick the ball in the 45 to 90 degree angle from where it was facing very quickly. An attribute of this kick was that the dog didnt turn much at all, in fact the kick could be executed and the dog would finish the kick facing in the same direction as when it started. The kick would move about half a metre. This kick was not viable or have a purpose in the actual game at this stage but could be developed further in the future. Because of this the advantages and disadvantages of the kick were not thoroughly assessed. Instead, more effort was focussed on the medium and large turn kicks.

Initial Observations

It was observed that often the kick would work successfully for some time, then fail, then work again and vice versa. This at first seemed like irregular behaviour and that there was nothing that could be changed to make the outcome of the kick exectution more reliable. Closer investigation, however, revealed that the cause of this irregularity was not the actual kick parameters, but the starting state from which the kick is executed.

Discovery of the Success/Fail Condition

The underlying motion package called Pwalk allows input to be taken and used for the legs every half step, or half revolution if you like. At the midpoint of the step the legs are in one of two positions. The left leg is up and the right leg is down, or the right leg is up and the left leg is down. The first of these positions in shown in Figure 6.9.

As a result of the different starting positions of the legs when the kick is engaged, it was observed that the kick was successful or it failed according to the position of the legs. As further observation and investigation proceeded the reliablility of different angle kicks, with different leg starting locations was also observed. What had been the observed irregularity before was really just the irregularity of the starting leg for the kick.

For the medium kick (90-120 degrees) the successful execution start location is where the dog has the *same* leg raised as the direction in which it is going

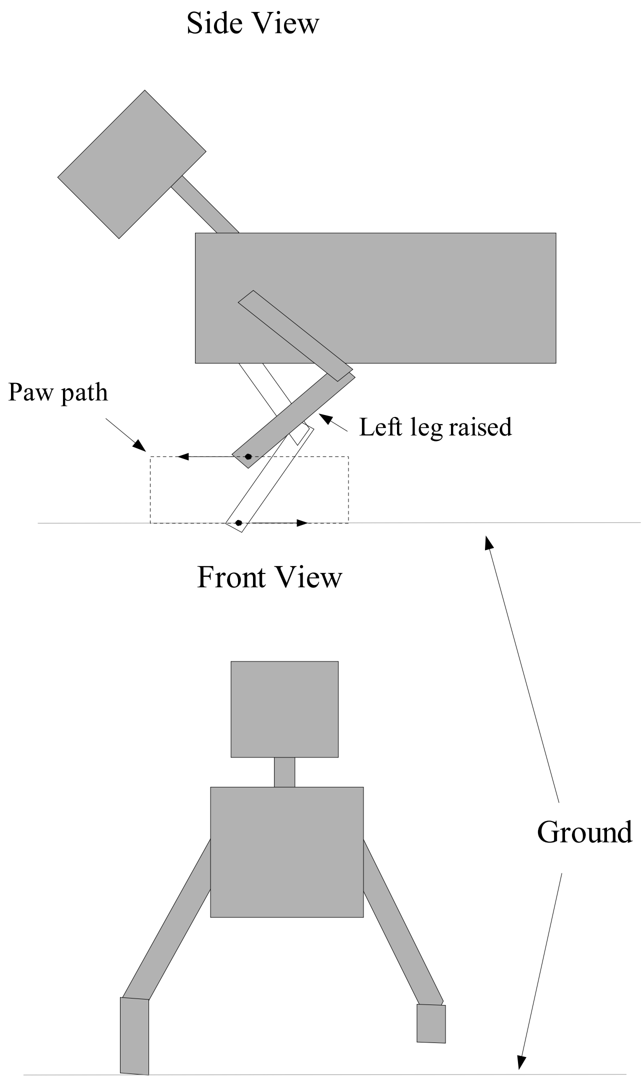


Figure 6.9: Front and Side View of a leg starting position.

to kick, e.g. the dog is kicking right, so the *right front* leg should be raised before the kick execution takes place. For the large kick (120+ degrees) the successful execution start location is where the dog has the *opposite* leg raised as the direction in which it is going to kick, e.g. the dog is kicking right so the *left front* leg should be raised before the kick execution takes place. For the small kick (45-90 degrees) it is the same as the large turn kick for the successful execution front leg start location.

The use of the turnkick with the correct starting locations for the legs increases the reliability of the kicks dramatically, the kick was now quite successful under the testing circumstances.

6.9.3 Adjustments to Kick Execution

You can modify the number of turn cycles in the kick execution to change the direction in which the ball will be kicked. But there is also another attribute which can be adjusted to the modify the kick and will need to be adjusted depending on the circumstances in which the kick is being used. This attribute whether or not to grab the ball before the kick, and if the dog is to grab, at what point in procedure of approaching and kicking the ball should it be grabbed.

Grabbing the ball ensures that the it is in the correct position before the kick is exectued and increases the reliability greatly. The reason for this is that when the dog is approaching the ball to get it between it's front paws, it is moving forward. Now, when the turn kick is executed without the grab the forward movement is stopped and as a result the ball will still have a forward motion component in being pushed by the dog's chest. This can often cause misfiring of the TURNKICK as was described above in section Background, where the ball comes out from between the paws prematurally.

If indeed it is decided to grab the ball before the kick to make it more reliable, there is still the need to set the point at which the ball will be grabbed. Grabbing the ball later, that is when the head tilt angle is less, will increase the reliability of the kick. Grabbing the ball earlier will reduce it.

Problems

If any of the turn kicks were used in play where the dogs were in a scrum. If a dog tried to do a turn kick it would not really be able to move. Because it didnt turn, it was most likely to be the case that the ball was still stuck between the front paws. The motion of the dog during turn kick did not make it obvious to an onlooker that the dog was trying to get rid of the ball. Because of this, it was questionable as to whether the Ball Holding Condition was broken or not and as a result the dog may be penalised for Ball Holding even though it had tried as was not able to eject the ball.

6.9.4 Conclusion

Three kicks were developed but in practice only one of the kicks was used. The large turn kick would often kick the ball too far around. This would occur because of the procedure in which the dog would approach the ball. It was found that is was quite unreliable and possibly dangerous for a dog to approach the ball and attempt to grab the ball when it was facing towards its own goal. For safety reasons, especially in the context of a game the approach ball procedure was modified so that the dog would only approach the ball to grab the ball when it was not facing its home goal. If the dog now tried a large turn kick in this circumstance the ball would go too far around and the kick would not have been as successful as if the medium turn kick would have been used. As discussed earlier the small turn kick was not ready to be used in a game situation.

The full potential of the reliability of the kick was not used in the final game strategy. To get the right condition for the most reliable execution of the turn kick there were setup costs. The dog's legs needed to be in the right position or it would have to wait an extra Pwalk half-cycle for them to get there. This was disrupting to the flow of the dog's movement, and there needed to be more testing to get this part of the kick, the shuffle, into a more workable state. Even though this aspect of the kick was not used, the kick is still more reliable than last year because of the more extended front legs and the faster speed at which the turn is made.

As development of the approach ball routine proceeds and the dogs become more 'coordinated' in close-ball interaction, these kicks could be used with a much greater effectiveness than the current system allows, saving a large amount of time that is wasted coming around the ball to approach it from the correct side.

Chapter 7

Strategy

Section author: Zheng David Wang and Timothy Tam

7.1 Abstract

This report will cover the details of game play strategy in RoboCup 2002. Strategy is composed to small modular behaviours that can be easily reused. This report will iterate through the behaviours and describe their task, purpose and usage. A top down approach will be taken. The high level strategy will be introduced first that represents the general tactics used in game play. Then, tasks such as getting behind the ball and positioning are described. Finally, the simple tasks or skills such as getting towards the ball, kicking the ball, etc will be addressed.

7.2 General Strategy

This outlines the general behaviour of the robot. It determines the category of actions the robot should be choosing based on the current situation. This is presented in the flow diagram in Figure 7.1. There are three main groups of actions: Positioning, Ball Approach, Close Actions. They are highlighted bold in the flow diagram.

Processing in the robot begins when the camera captures an image. The vision module first analyses the image to identify the objects and produces the vision model. The localisation module then uses those information to update the world model and the location of the robot in the world. Finally, the strategy code is called. This is the entry point to the flow diagram.

To begin understanding the general strategy, the idea of an **atomic action** must be introduced. An atomic action is a strategy level task that is different

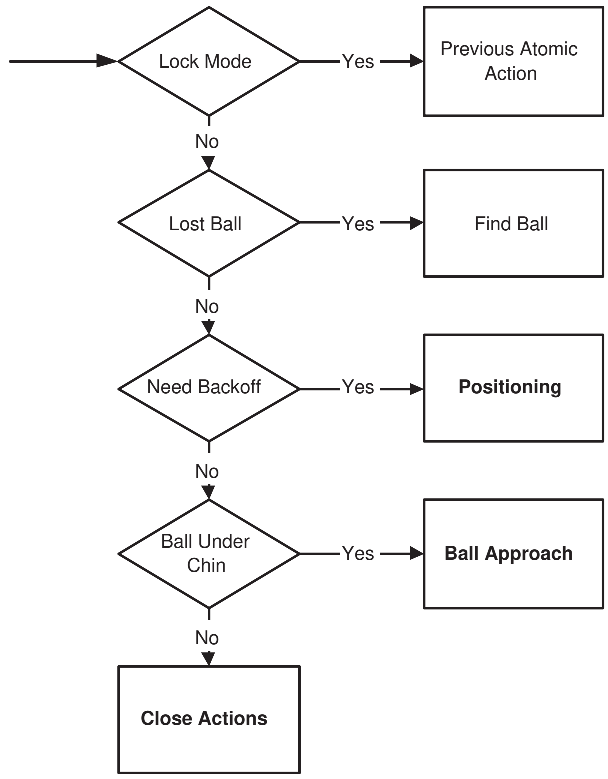


Figure 7.1: General Strategy Flow Chart

to actions defined in local motion module such as walking or kicking. An atomic action is a higher level action that requires a sequence of basic local motions to be executed. During the executing of this task, the robot should not make other strategy decisions. For example, in the grab ball routine, the robot first stretches out its arms to capture the ball. Then, it lifts its head to look ahead for a clear kicking path. Since the robot is unable to see the ball when it lifts its head, if it enters back into the strategy decision tree to choose an action, it will execute the find ball routine instead of the kick we want. This means the robot needs to be locked in an atomic action that grabs the ball. An atomic action either successfully completes or an exception happened. An exceptions is an unexpected events such as a ball popping out when we try to grab it.

At the beginning of the general strategy, the robot first checks if it is locked in an atomic action. If the flag is true, then it will return to the current atomic action. Next, the robot will check if it has lost the ball. This happens when the ball has not been seen in the previous 6 camera frames or the robot is currently trying to find the ball and it has not seen the ball continuously for 2 camera frames. If these conditions are met then the robot executes the find ball routine.

```
If ( (last time saw ball > 6 frames) ||
     (finding ball && saw ball < 2 frames) ) {
    FindBall()
```

```
}
```

The next condition in general strategy checks if the robot needs to back off a team-mate. This condition is met if the robot is confident that a ball is within 50 cm and a team-mate that is close to the ball is ahead of it. Another way the condition can be met is if there is another team-mate right in front of the robot and within 60 cm.

```
If (
    (confident(ball) && ball.dist < 50 && confident(team-mate) &&
        ( team-mate.dist - ball.dist < 50) ) ||
    (team-mate.dist < 60) ) {
        BackOff()
}
```

The finally condition in general strategy checks if the ball is under the chin. A robot verifies this by checking the value of its head tilt. The robot will execute close ball actions such as grab or kick if this condition is true. Otherwise, it will simply approach the ball.

```
If (head tilt < -55 && abs(head pan) < 15 && ball.dist < 20
    && last time seen ball < 6 frames) {
    CloseBallAction()
} else {
    ApproachBall()
}
```

7.2.1 Tests

The strategy is tested by running weekly games with a full team. The newly developed code is competed with the last version. This tests that the new version has made improvements over the last version. Because of the stochastic nature of the game, at each run we look for improvements to the problems that we observed in the last run. At the same time we identify new problems. We look for the general control of the ball between the team. We also keep statics on the percentage of time the ball spends on each half of the field. We also keep track of the scores over the weeks to ensure progress.

Anther type of test we conduct is to test the speed of the robot in scoring a goal. There are 3 tests we conduct. The first test requires a robot to score against an empty goal. The second test requires a robot to score against a goalie. The third test requires 2 robots to score against a goalie. In these tests we first remove the ball from the field which places the robots in find ball mode. Then, we place the ball at the upper half of the field, at 1/4 of the way across the field, at 1/2 of the way across the field and at 3/4 of the way across the field. In these tests we record the time taken to score.

7.2.2 Observation / Short Comings

When testing the strategy, a number of observations were made. During early stages, the robot had a reticent behaviour towards the ball due to the positioning behaviour. We also discovered problems like the dogs been too cluttered or too spread apart. These will be discussed further in the section "Positioning".

When running the scoring speed test, we discovered that a robot may be excellent in scoring alone. It can easily score 10 goals in 5 minutes. However, when a goalie is placed there, the robot is unable to score 10 gaols with in 10 minutes. From this observation we found that the set of actions to score against an empty goal is different compared to when a goalie is present, because a robot that is able to score against an empty goal can simply nudge the ball slowly into the gaol. This is not true when a goalie is present. From the tests we also found, when 2 robots (forward) tries to score against the goalie, the location of second robot is crucial. When the robot with the ball is attacking the goal, if the team-mate is on the opposite side of the goal, it is often able to recover the ball before the goalie.

7.2.3 Improvement

Seeing the initial short comings of the strategy, force field was introduced to solve the problem of robot positioning. Force field can be based on the robot's world model or its wireless model. Also, to ensure a robot is able to score even with the presence of a goalie, fast accurate actions were incorporated into the strategy. Moreover, the back off strategy is improved to ensure better robot positioning for a fast attack and retrieval.

7.2.4 Roles

In the strategy, there are two roles, an aggressor or an assistor. An aggressor will actively attack the ball while an assistor will assist in the aggressor by positioning in the right position. Role determination decides the role of the robot by comparing their believed distance to the ball.

The aggressor is the robot that believes its distance to the ball is the shortest in comparison to all other team-mates beliefs of their distance to the ball. From random errors in vision and delays in wireless, it is possible for robots to all believe they are all aggressors or assistors. Taking into account that if all robots are assistors then no one will go for the ball, it is rather desirable to have all robots as aggressors. This is achieved by having a threshold in the distance comparison. A robot will become an assistor if its distance to the ball is more than the distance between the ball and closest robot to the ball plus the thresh hold.

In the final strategy, this technique was actually not adopted, because via experiments we found the robots were too scattered and usually only one robot attacked or defended at a time. Hence, this was abolished and the robot is set to aggressor by default and they only got out of the way when needed.

7.2.5 Desired Kicking Direction (DKD)

During game play when the robot decides to kick the ball or get behind a ball, a general guidance must be given in the direction to execute the actions. This guidance is given in the form of a graph that provides indications of the general flow of the ball. Using the graph in figure 7.2 the robot is given an angle anywhere on the field indicating the direction it should kick the ball relative to its current heading. This angle is called the Desired Kicking Direction or in short DKD. Also, when the robot tries to get behind the ball, it will rotate behind the ball lining itself up with the DKD.

The aim of the strategy is to get the ball up the field as fast as possible, hence the flow of the ball is away from its own goal and into the opponent's goal. When the robot is at the upper half of the field, it will try to kick the ball into the opponent's goal. If the ball is on its own half, it will try to clear the ball into safety by kicking it away from its own goal. Finally, when the ball is along the edge of the field, the robot will try to slide the ball along the edge of the field. From experiments we found by sliding the ball along the edge using a turn kick, the ball can travel much further. Dribbling the ball along the edge also has the advantage of avoiding opponents because they normally don't come along the edge of the field.

7.3 Positioning

The choice of positioning for the forwards is important for attack and defense. Ideally the forwards should be spread well enough over the field so that a lost ball can be easily found. Furthermore if one forward has the ball and is tackled, another forward should be positioned nearby in support for quick retrieval of the ball. It was found through experimentation that one of the greatest degradation of play was when the dogs attacked the ball at the same time and collided with each other. The positioning of dogs avoids this problem at all costs.

7.3.1 Visual Back Off

When the robot detects through its vision model that there is a team-mate extremely close to itself (utilTeammateInFace()), it steps back. Initially one might think that this would cause two dogs approaching the ball to both simultaneously step back, thus reducing the aggressiveness of play. However in practice only one robot will reach the ball first. This is because, as a robot get close to the ball, all it sees is the ball. Hence, the closer robot will only see the ball whilst the team-mate will lose sight of the ball (obscured by the closer robot) and see its team-mate (the closer dog) in its vision and step back.

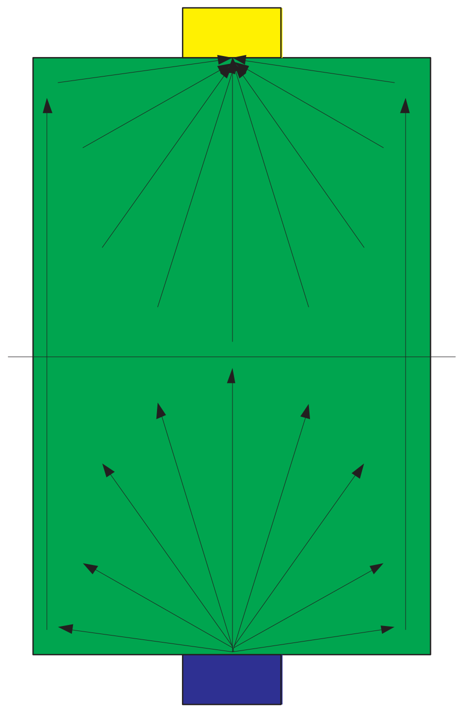


Figure 7.2: Desired Kicking Direction (DKD)

In game play it was found that this positioning was used regularly to prevent dogs simultaneously attacking the ball. Furthermore, when the team-mate steps back, if the ball were popped out due to a fumble or tackle, it was in an excellent attacking position to retrieve the ball.

Visual backoff has the highest precedence over the other positioning techniques (force field and wireless positioning). This justification for this is that the vision system always contains the most up to date information and if it detects that there is a team-mate extremely close by, the only logical move is to step back rather than using any other fancy positioning technique.

7.3.2 World Model Rotational Backoff

This is triggered when a dog is close to the ball (within 50cm) and it calculates through its world model that there is a team-mate closer to the ball. The dog will rotate as quickly as possible into the position that is directly behind the ball facing the opponent goal. The direction the dog rotates is relative to the dogs position on the field or its DKD as shown in the diagram below.

The dog on the left rotates counter-clockwise to align itself with the DKD whereas the dog on the right rotates clockwise. If the dog is close to the left wall then it must rotate clockwise and conversely if the dog is close to the right wall it must rotate counter clockwise. Otherwise, if the DKD is positive the

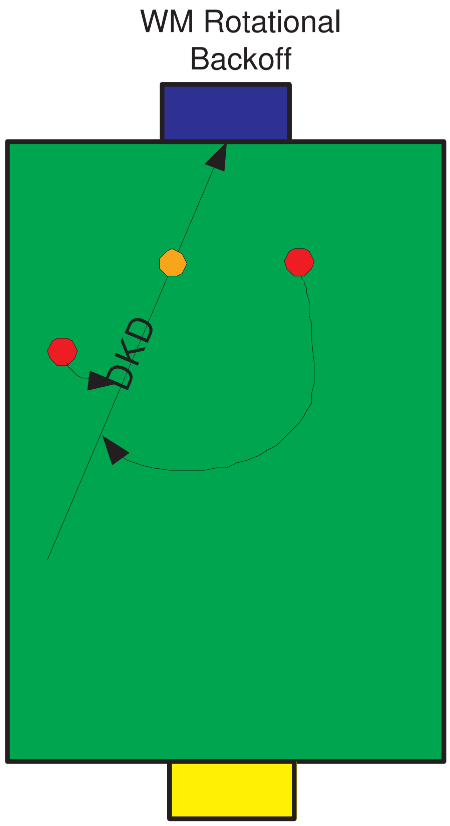


Figure 7.3: World Model Rotational Backoff

dog rotates clockwise and conversely if the DKD is negative the dog rotates counter-clockwise.

If the dog is not in an attacking position the rotational movement is locked for 20 frames. Otherwise if the dog is in an attacking position then the rotational movement is not locked at all. A dog is in an attacking position when `utilIsUp()` is true, which checks whether the absolute value of the DKD is less than 60 degrees. This additional clause was added because there are certain situations where it is suboptimal for the closest dog to the ball to attack it. For example consider the situation (see figure 7.4) where the team-mate (a) is the closet to the ball and it is facing its own goal. The other team-mate (b) is also close to the ball but facing away from its own goal. In this situation team-mate (a) should backoff and allow team-mate (b) to attack the ball. The above condition allows this by locking (a) in the back off whilst not not locking (b). (b) will attack the ball as soon as team-mate (a) rotates out of its vision.

The destination rotational position of being behind the ball and facing the opponent goal is chosen for defensive reasons. It was found through experimentation that it was better for a dog to backoff behind a ball so that it can quickly retrieve a loose ball and to obscure an opponent attack.

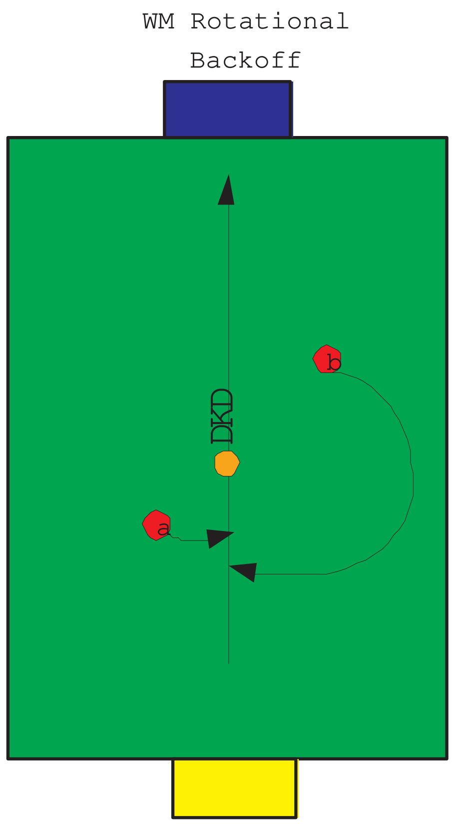


Figure 7.4: World Model Rotational Backoff Example

It should be noted that the world model information regarding team-mates is extremely noisy. Thus, the calculations are often inaccurate. This can cause both dogs believing the team-mate is closer to the ball. This will result in both robots executing rotational positioning. However, in practice, one dog will loose sight of the other dog first allowing it to attack the ball while the other dog maintains the rotation.

This world model rotational backoff is at one precedence level lower than the visual backoff.

7.3.3 Force Field

Force field is a technique used to keep the forwards uniformly separated over the game field by placing forces of repulsion on each dog and forces of repulsion from the edges of the field as shown in the diagram in figure 7.5.

The magnitude of the force is $\frac{K}{d^2}$ where K is a constant and d is the cartesian distance between the dog and a repulsion object. For a given forward whose position is (x1, y1) the repulsion objects are (a, b) (where (a, b) is the po-

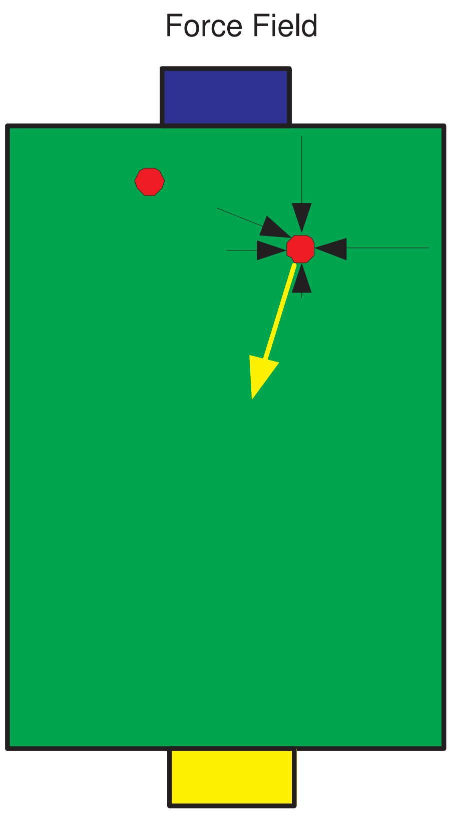


Figure 7.5: Force Field

sition of the other robot(s)), (0, y1), (FIELD_WIDTH, y1), (x1, 0) and (x1, FIELD_LENGTH). A vector addition is performed on all the forces and the resultant force is shown in yellow.

From the formula it can be easily seen that the magnitude of the force is inversely proportional to the dog's distance to the repulsion object. Hence the closer a dog is to a repulsion object the quicker it moves away from it.

Force field positioning was initially developed to solve a forward clumping problem (where all the dogs would go for the ball and get in each others way). Upon implementation it was found that it was an excellent technique in keeping the dogs spread. However the accuracy of this positioning technique greatly relies on the accuracy of a dog's knowledge of its team-mates' positions.

It was found that the world model data tends to be too noisy, and caused the forwards to be too spread when attacking the ball (there was no form of assistance to an attacking team-mate at all). As a result this positioning technique was replaced when visual backoff and world model rotational backoff was developed. However it is still used in wireless positioning (see below) since the team-mate positional data is highly accurate when the wireless network is up. It is also used to keep the dogs spread during the find ball routine.

The force field model used is quite primitive. Improvements can be made by having additional features such as an attraction on the ball (to reduce the spreading of the dogs), repulsion forces from opponents and attractive "hot spots" on the field that are deemed to be strategic positions i.e. in front of the goal when a team-mate has the ball.

7.3.4 Wireless Positioning

Wireless positioning is the most accurate of all the position techniques discussed as it directly uses the team-mate's belief of their own position rather than the dog's vision information about its team-mate.

There are three forwards in a team. The dog that is the furthest from the ball out of all three forwards positions itself using the force field model. The dog that is the second closest to the ball out of three forwards positions itself in a wing position to the team-mate closest to the ball as shown in the diagram in figure 7.6.

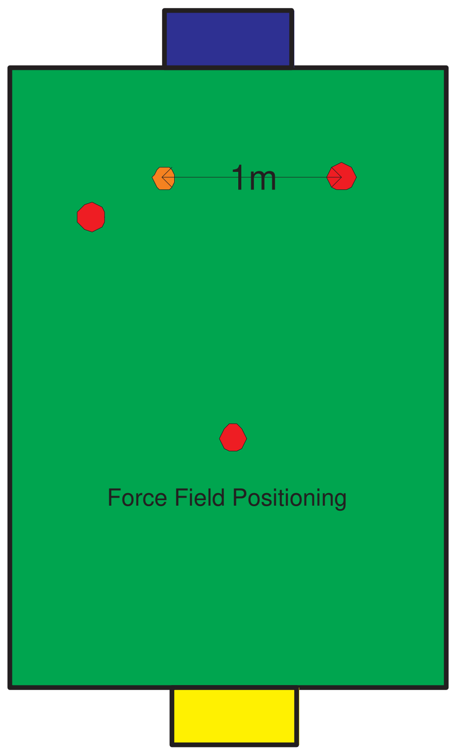


Figure 7.6: Wireless Positioning

If the position of the dog closest to the ball is (x, y), the wing position of its team-mate is (x+100, y) when the ball is in the left half of the field and (x-100, y) when the ball is in the right half of the field. The x coordinate of the wing position is capped to ensure that the team-mate remains on the field.

The general result of wireless positioning is that one forward is attacking with the ball with a team-mate assisting in a wing position, ready to retrieve any balls that pop out. The remaining team-mate is hovering behind the other two dogs in a defensive position ready to prevent a possible attack and prepare to retrieve a ball that may happen to roll out down the field. The choice of positioning can be summarised in the figure 7.7.

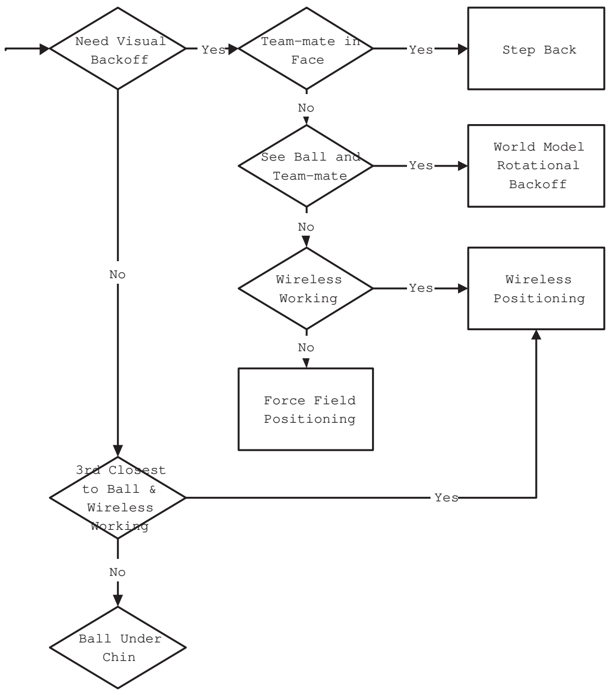


Figure 7.7: Wireless Positioning Flow Diagram

7.3.5 Tests

Initially to verify each positioning strategy, two dogs were placed on the field and the ball was moved manually around the field. The movement of the dogs relative to each other were record. This test was then repeated with three dogs on the field and finally during a full game.

To test Force Field Positioning and Wireless Positioning a dog's world model was sent to debugging software and observed. The effect of a dog's misperception of its team-mate's positions on positioning was noted.

7.3.6 Observations / Shortcomings

The positioning strategies which was most effective in game play were Visual Backoff and World Model Rotational Backoff. Force Field and Wireless Positioning strategies work better in stand alone testing. This is because in a game to the number of robots on the field increases the noise in the robot's localisation system and hence reduces the effectiveness of more complex strategies.

Furthermore wireless positioning was not even used in the competition because increased processing caused by sending wireless data caused a reduction the robot's reactions. This resulted in less cooperation between the dogs and due to the larger size of the field and the ball often being obscured by robots, there was often at least one dog spinning around executing the find ball routine, not doing anything useful. This didn't cause problems in the early rounds of the competition when a single forwards could score goals on its own due to weaker opponents. However the lack of team cooperation was costly when we played a decent team. When the ball was stripped from our attacking robot there was no robot in support.

7.4 Ball Approach

In the strategy there are four difference ways to approach the ball. The choice is based on the position of the dog.

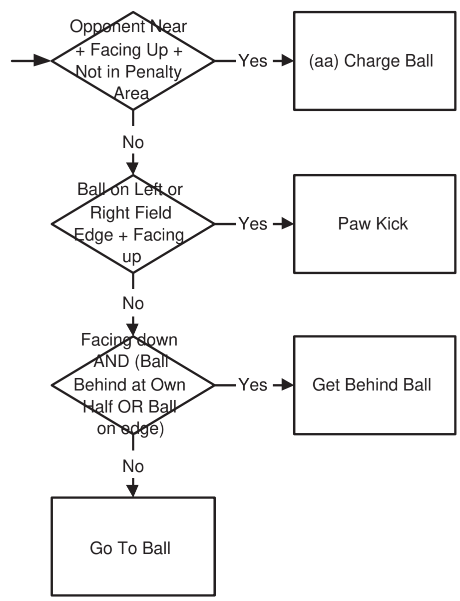


Figure 7.8: Choice of Ball Approach

7.4.1 Got To Ball

This is the default method to approach the ball. The dog walks directly to the ball adjusting its forward, left and turn parameters based on the ball heading and distance. This is identical to the previous year's code. The robot uses zoidal walk by default to minimise the time taken to approach the ball. However the robot switches to normal walk if the turn value is too large to prevent instability and also when the ball is within 20cm to prevent fumbling. As soon as the normal walk is activated, it is locked for a certain period of time to avoid instability due to switching between zoidal and normal walk.

7.4.2 Get Behind Ball

This routine tells the dog to move to the position directly behind the ball facing the DKD. This is activated on two occasions: when the ball is on the edge wall with the robot not facing up and when the ball is inside the defensive half.

The routine is activated in the former case to avoid the risk of fumbling the ball into its own half because, attacking the ball along the edge will cause the ball to role in the attacking direction. Also, it is difficult to grab the ball or turn with the ball when you are not close in 90 degrees to the wall.

The routine is activated in the latter case to minimise the chances of pushing the ball further into the dog's own half and more importantly to minimise the chances of scoring an own goal. It is true that approaching the ball from behind wastes more time but, from practise matches it was found that this safety precaution was necessary to remove costly fumbles.

7.4.3 Charging

Charging was discovered during the development of a new walk –Zoidal Walk. This is a faster walk that brings the front paws of the robot closer together. Unfortunately, this walk is less stable than the previous walk and tends to kick the ball out because the front paws are too close. However, we have turned this weakness into an extra skill called charging, where it exploits these properties uses it to charge at the ball to knock it out of opponent's hand.

Charge ball is identical to the go to ball routine except the dog does not switch to normal walk when the ball is within 20cm. This routine is called when the dog is facing up (absolute value of DKD is less than 60 degrees), it is not in the opponents penalty region and there is an opponent nearby. The logic if an opponent is nearby there is a great chance that the dog will be tackled and thus loses the ball if it slowed down to trap the ball under its chin. Instead, it chooses to simply charge at the ball which will knock the ball in the forward direction. This favours the attacking dog as it is facing the right direction whereas the opponent dog has to turn around and chase. Furthermore as UNSW 2002 had a speed advantage compared to other teams this strategy turned out to be very effective in competition.

7.4.4 Paw Kick

Paw kick is a routine where the dog tries to aim either it's left or right front paw at the ball on approach. It is a method to propel the ball in a forward direction

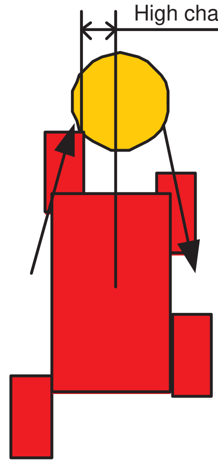


Figure 7.9: Charging

without requiring the dog to slow down. In strategy this was only called when the ball was on the edge of the field and the dog was also on the edge and facing the attacking half. This allowed the forwards to quickly push the ball up the field when it was on the edge.

7.4.5 Tests

Initially the ball approach strategy was much more complex. The field was divided into many more subregions and a ball approach was picked based on the region the ball was and the direction the dog was facing. However this strategy was ineffective as inevitable errors in localisation caused the dog to "jump" regions causing the dog to fluctuate between decisions. This behaviour caused jerkiness in the robot's movement and led to the undesirable result of ball fumbling.

To verify the correctness of the ball approach strategy, a robot was placed on the field while the ball was continuously moved around and placed in tricky positions e.g. the corner of the field. Tests were performed with the dog's view obscured in order reduce localisation and the resulting behaviour in strategy was observed.

7.4.6 Observations/Shortcomings

Because of the way the paw kick was designed it cannot be locked into execution once it is called. Furthermore upon execution of the paw kick, if the robot determines that it is out of line with the ball it will strafe in an attempt to come to line with the ball again. An undesirable behaviour that occurs on rare occasions, due to mislocalisation the robot, caused the robot to strafe away from the ball or fumble it.

7.5 Close Actions

7.5.1 Got Ball

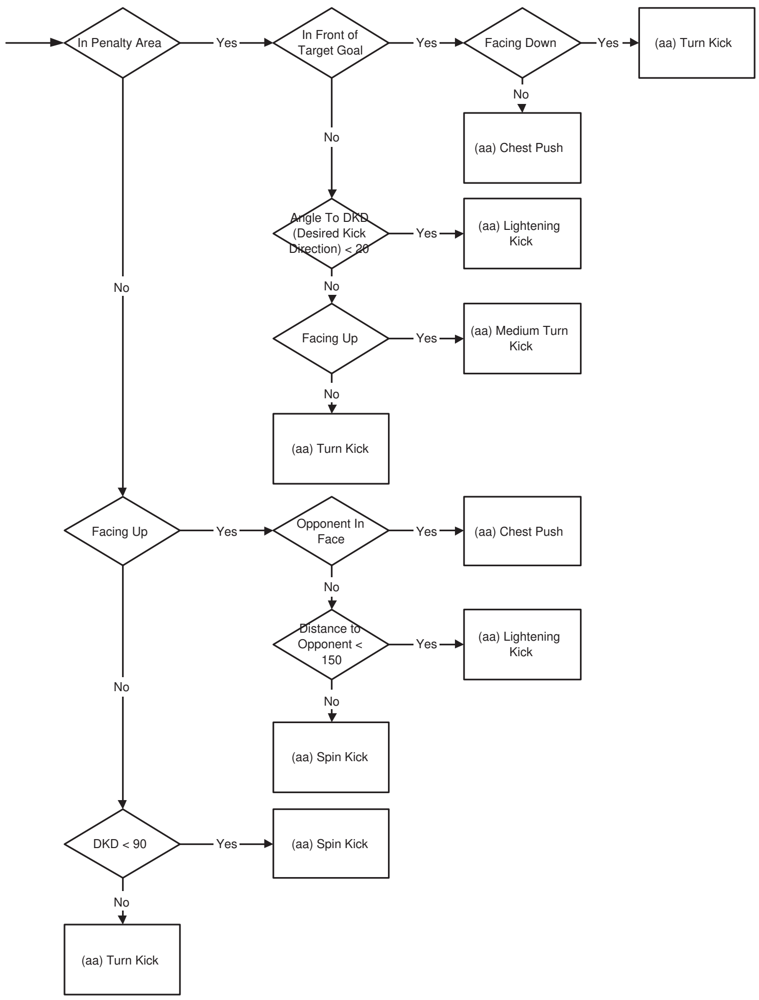


Figure 7.10: Got Ball Flow Chart

This decides the robot's action when the ball is right under its snout. The condition for determining the ball is under your chin is discussed in the "General Strategy" section. The idea behind got ball is to get the ball up the field as quickly as possible and then do fine actions only near the goal.

When the robot is in front of the goal, it will perform actions that have greater control on the ball. At this point it will execute a spin chest push if it's not facing down according to DKD otherwise, it executes a turn kick. Both of these actions have a low risk of loosing the ball. The spin chest push actually grabs the ball and spins the ball around you, taking full control of the ball, then it executs the chest push action. Spin chest push is not executed when you are facing away because it may exceed the 3 second ball holding penalty. The pseudo code is show below.

```
If (in font of goal) {
    If (abs(DKD) > 120) {
        TurnKick()
    } else {
        SpinChestPush()
    }
}
```

If the robot is not in front of the goal but at the upper half of the field, the general strategy is to shoot for the goal as quickly as possible. In this case, the lightening kick is used when the robot is aligned with the goal according to DKD. If not, a medium turn kick is used for small angle deviation and a normal turn kick for a large angle deviation. The pseudo code is show below.

```
If (in penalty area && !(in front of goal)) {
    If (abs(DKD) < 20) {
        LighteningKick()
    } else if (abs(DKD) < 120 ) {
        MediumTurnKick()
    } else {
        NormalTurnKick()
    }
}
```

When the ball is located far from the target goal, the objective is to get the ball away from our own goal as quickly as possible. This situation only requires rough actions to be executed and we have more time to complete the action because, the actions can withstand bumping from opponents. Under these conditions the robot will first try to execute a spin kick. This occurs if it does not sense any near by opponents. Otherwise, it tries to get rid of the ball quickly using a lightening kick so a scrum does not result. To get out of scrums the robot use a chest push. This is detected if it sees an opponent robot right in front of it. The pseudo code is show below.

```
If (!(in penalty area) ) { If (abs(DKD)  < 60 ) {
    If (opponent in front of you) {
        ChestPush()
    } else if (opponent in font < 150 cm ) {
```

```
        LighteningKick()
    } else {
        SpinkKick()
}
    } If (abs(DKD)  < 90 ) {
        SpinKick()
    } else {
        NormalTurnKick()
    }
}
```

7.5.2 Spin Kick

Kicking capabilities is especially important in the robot soccer games. Having a reliable and powerful kick is crucial, but we must utilise every opportunity to execute this skill. In previous years the robot had to get a round the ball in order to execute a kicking action. This often takes too long and gives opportunities for opponents to retrieve the ball while the robot is getting behind it. The solution is the have full control of the ball while the robot is getting behind it.

The get behind ball routine developed last year revolved around the idea that, the ball is stationary and we can't touch it while we are moving into position. This because we are clumsy and we will knock the ball out of our control if we get too close. In this routine we rotate around ball. What about making the ball rotate around us?

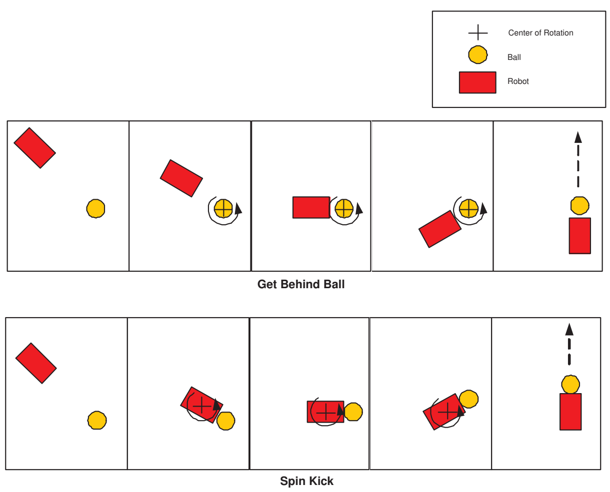


Figure 7.11: Comparing Get Behind Ball and Spin Kick

During the development of the turn kick last year, we found it was very easy for the ball to get trapped under the robot's snout when executing the kick. Looking at the behaviour of this turn, we realised the dog had an ability to turn with the ball without losing control of it. This is a very aggressive movement where you can almost steal the ball of the opponent and run with it, robbing the opportunity of the opponent to even touch the ball. In addition, since we are holding the ball, we can execute a kick immediately after the spin. Hence, we derived the name Spin Kick.

Spin kick allows the robot to come in any direction of the ball and kick it in any desired direction. This is because spinning and kicking very fast movements. After a number experiments, we have found the robot is able to perform a 180 degree Spin Kick under the 3 second ball holding penalty.

7.5.3 Tests

The first test we conducted had the ball already trap under the robots snout. This robot is then made to spin on the spot and try to continue holding the ball. Under this test,it allowed us to tune the turning parameters to obtain the maximum turning rate without loosing the ball. The parameters we found are as follows.

Parameter	Value
fsO	25
ffO	80
forward	1
turnCCW	35
pan	0
tilt	-50

The parameters tabled gave us 100% reliability. Under many tests, we found the robot can turn faster with a maximum turnCCW of 45 degrees. However, this caused the legs to slip on the carpet more and made the odometry information from the local motion module unreliable.

The second test we conducted involved placing the ball at a stationary location then requiring the dog to run up to the ball and execute the whole spin kick sequence. The purpose of this test is to verify the reliability of the kick and test the maximum angle the robot can spin under the 3 second ball holding penalty mark. Using this test we found that the robot is able to execute the spin kick successfully almost all the time, provided we don't fumble the ball during ball approach. The robot is able to execute a full 180 degree spin kick between 2.7 to 2.9 seconds. In addition, the spin kick is successful even when the ball is on the side wall.

7.5.4 Observations Short Commings

There are a number of disadvantages of this action. First, during the spin, because the robot looks down at the ball, it has to rely completely on world model. With a good world model, the robot is able to spin to the correct direction and score. However, during a scrum, the robot often won't be able to spin exactly to where it wants. This can make the kick following the spin a lethal action if the robot is pushed by the opponent to face its own goal.

The second disadvantage of Spin Kick is that it takes almost 3 seconds to execute a 180 degree turn. During experimental games, we have found this period is too long. By the time you have turned the opponents are already in front of you and blocking your way.

The final disadvantage of this kick is that, it is highly dependent on the physical environment, i.e. the carpet and the robot. This is because during the turning phase the robot can experience dissimilar slippage on a different carpet that will cause errors in turning angle. This problem can be solved by turning the kick parameter for different carpets.

7.5.5 Conclusion

Over all Spin Kick is a very powerful, fast, aggressive and reliable action. However, caution must be taken in deciding when to use it as it is heavily dependent on world model accuracy. Moreover, the effects of physical environment should be understood.

7.5.6 Spin Chest Push

This action is exactly like Spin Kick except we execute a Chest Push at the end of the spin. The advantage of this this skill is it allows the robot to push the ball into the opponent goal even when a goalie may be present. The other advantage is that if we make a mistake in the kicking direction, we can easily recover the ball. This is because the chest push has a short kicking range. We need this fast recovery especially when we are right in front of the opponent's goal.

7.5.7 Other Kicks

There are a number of other kicks available listed as follows:

- Turn Kick
- Normal Turn Kick
- Medium Turn Kick
- Goalie Kick
- Lightening Kick

They are very simplistic that does not involve any strategic decisions. For this reason I refer you to the "Local Motion" chapter more more details.

7.6 Other Developments –Simple Strategy

Section author: James Brooks

It was decided to start on a completely different type of strategy for gameplay. The reason for this was that we did not want to be continually trying to improve a sub-optimal strategy. The hope was that we may increase the success of our main strategy to beyond the maximum of the current path by finding a global maxima, or just a greater local maxima. This would be through development and discovery and then reintegration back into a single main strategy for our dogs to be used in the competition.

In development of strategies it is good to be able to test it against other types of strategies other than against itself, or continually against on particular type of strategy. Testing against different types of strategies will reveal different weaknesses in your own strategy. A secondary aim for producing this strategy was so that testing of the current strategy would be more extensive and more helpful for further development.

7.6.1 The Strategy Itself

The strategy that we were using was a complex one, I will refer to this strategy as 'the current strategy' or **C**S. To combat the complexities of the CS, we decided to employ a 'simple strategy' or **S**S. The strategy we started with is listed below.

Strategy

1. **I**F lost the ball THEN findball gravitating towards the home position
2. **I**F found the ball **T**HEN Approach the ball
3. **I**F close enough to grab the ball **T**HEN Grab the ball
4. Pan the head from left to right, or right to left looking for a goal, whether target goal or own goal
5. IF we see target goal **T**HEN check angle,
   - **I**F distance small **T**HEN chest push
   - **I**F angle small **T**HEN align and front kick
   - **I**F angle large **T**HEN do a turn kick
6. **I**F we see our own goal **T**HEN do a turn kick in the appropriate direction
7. **I**F we see no goals **T**HEN use world model information, and turn kick towards target goal

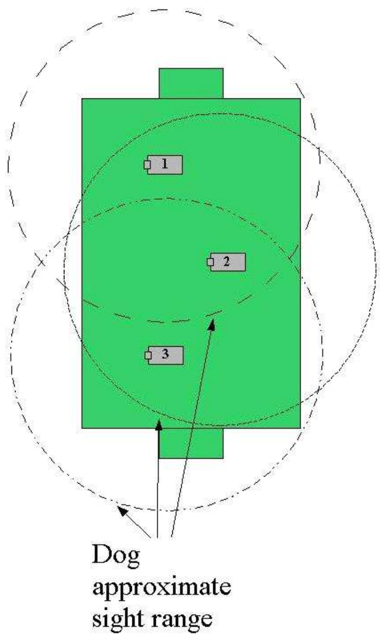


Figure 7.12: A staggered formation provides better coverage of the field.

Now to explain what this means. The strategy code is continually called each time the dog is ready to make a decision. It works as a series of conditions, somewhat like a tree. The result is that each successive level in the 'strategy tree' has, as satisfied pre-conditions, all the previous steps. For example, when we were grabbing/holding the ball the dog knows where the ball is and it has already approached it, and when the dog is about to kick it knows that it has already grabbed the ball etc. This process simplified the coding and debugging enourmously as we will see later.

Now some terms will need to be explained. The 'home position' is a position on the field that each dog is assigned to. These can be worked out when the dog first starts the match or optionally hardcoded into the strategy. These home positions were chosen such that when all three of the dogs (not including the goalie) were on the field, the ball could be anywhere on the field and at least one of the dogs would be able to see the ball easily and the field would be covered in the best possible way. The way chosen to cover the field sought to maximise the area on the field that was covered by more than one dog. This is illustrated in Figure 7.12. The other terms should all be described in other parts of this report.

7.6.2 Preliminary Results

In the testing of the SS a few observations and characteristics were noted. Some of were advantageous to the SS over the CS, others were drawbacks from the CS. I will proceed through them one by one.

Vision Model Usage

The SS relied heavily on the Vision Model data (this data comes direct from the vision into the variables such as vtGoal, vBall, voGoal etc) and this information, when available, was given priority over other kinds of information such as the World Model data. This brought solutions to a number of problems that had existed in the CS at that time. The CS sometimes caused the dogs to get confused about where the ball was because it was using the World Model data instead of the Vision Model data. Because the World Model data is built up over time, where fast movement of the ball pccurs it takes some time for the World Model data to catch up.

Another problem that it reduced was the response of the dog to the ball. The CS was much more based on the position of the dog when it had the ball and what to do in different positions. This would mean that if the dog didn't know where it was then it would have to look where it was before it kept going. Now no matter what kind of strategy that the dogs are playing it still is important for the dogs to know where they are to some extent, but with the SS this dependancy was greatly reduced. Looking back over the strategy above, the only point in which the dog needs to know where it is the case when it has looked and can't see any goals. Under these circumstances it is likely that the dog is a fair distance from the goals and consequently getting the right action quickly is generally not as crucial. What this meant in game was that first and foremost the dog would go for the ball then decide what to do. This provided some great advantages because the team was able to get control of the ball first which was a major factor involved in winning the game.

However there were drawbacks to using the Vision Model information as paramount. These drawbacks came in the form of seemingly erratic behaviour. This erratic behaviour was caused by the nature of the Vision Model data. Vision Model data is noisy. It is by nature noisy due to the fact that we are working with analog and highly variable data (light from the real world) being measured by limited accuracy instruments (cameras) and we are trying to derive precise information from the inputs, such as distances and angles, and all we have to work with is colour, which is highly dependent on the lumination of the objects, and blob position in the C-plane (see section X.X.X). Now one way to 'smooth' out this noise is to keep track of what is has been previously and change the data we use slower then what the inputs suggest, but this is along the same lines as how the World Model data is produced and will consequently lead us into the same problem.

Code Size

Having a smaller piece of code to deal with made debugging and modifying the strategy much faster and easier. Because we had the condition that as the program progresses through the strategy code each previous condition has already been met, it was very easy to find out the point in the program where an error occurs. For example you could stop to the program progressing to the next condition and so make sure that all the previous conditions and actions functioned correctly. The finite number of states that the program could be in also allowed testing of all the circumstances that could occur and thus making sure that the strategy would work properly.

This structure and the simplicity of the strategy also allowed easy comprehension for those who had not yet seen the strategy. This was of enormous benefit when it was undertaken to bring all the best components of each of the strategies together.

Ball Crowding/Hesitation

The SS was based around the individual dogs' interaction with the ball. In order to keep it simple and to keep the dogs on the most important task there was no conditions written concerning the dogs' interaction with other *teammate dogs*. This resulted in the dogs bunching up and crowding the ball because as soon as any dog saw the ball, regardless of whether there were teammates close to the ball the dog would go for it. This had effect of making dogs on the same team interfere with each other. For example if one dog was about to make a kick for the goal and another teammate sees the ball, be it between the legs of its teammate, it would come to attack the ball and score a goal for itself. This sounds disasterous but it was not so severe.

Using this technique in the SS actually somewhat fixed a problem that we were having with the CS at that time. The dogs of the CS were very hesitant when approaching the ball. This hesitancy was caused by the *BackOff Condition* being triggered. The *BackOff Condition* is a condition or test that, when triggered, causes the dog to alter it's course of action for a period of time because it believes that it's teammate already has control of the ball and it therefore does not want to interfere. The misfiring of this condition caused a fair bit of distress because often the dogs would have the opportunity for a clear break and potentially to score an 'easy' goal and then loose that opportunity because they would hesitate to the point that the other team would end up getting to the ball and controlling it before them.

Dog Flipping

The SS, being simple also didn't take into account any visual information concerning *opponent dogs*, so no matter where the opponent dogs were placed it would not really affect the decisions of the SS only in that the opponent dogs may block the view of the SS dog. At first glance this is a problem and indeed the problem only came to our attention through testing. The problem encountered was that when an opponent dog (it was inactual fact any dog but the problem was most often caused by opponent dogs) was directly in front of the kicking dog and the kicking dog was executing a *FRONTKICK*, instead of moving its arms down to the ground and forcing the ball out, the dog would have its paws caught on the other dog and end up *pushing itself up*. As a result of this the dog would become unbalanced and roll over onto its back.

When the dogs are on their backs they become useless as players until they get upright because they use the standard get up routine which will continue until the dogs are upright standing on all fours. This routine, and any way to flip the dog to the right way up, is and will be very costly in terms of time and in terms of control of the ball in gameplay, especially when the dog may be trying to clear the ball, or even when the dog is shooting for goal. When the dog falls over in these circumstances it is very likely that there are other dogs that are around it (we know that there will be at least one dog very close it because a dog does not just flip over by itself) and because the kick that it was trying to do would not have executed properly, the ball would be very close by. Because of this, often the dog will not be able to right itself for quite some time because as the dog tries to flip itself over, the other dogs will be in the way.

Ball Holding

One of the rules of the Sony Legged League is that the dogs are not allowed to *"Hold the Ball"*. For a definition of *Ball Holding* please see the notes explaining the rules of Robocup in section X.X. Now as this strategy developed it was recognised that there may be problems concerning ball holding because a large part of the strategy is carried out while the dog is holding the ball. Once the dog grabs the ball it then needs to work out where the goals are in relation to where it is now, line up with the goal if it can see them, then kick the ball towards the goal. This process can take some time because it may involve panning the head all the way from left to right at a speed where the dog can actually recognise the goal and then aligning with the goal while the ball is grabbed. These time consuming activites combined with the fact that often the ball is classified as being in the *Ball Holding position* for an amount of time before the grab is actually executed meant that often the dog would hold the ball for too long and be penalised for Ball Holding. Ball Holding carries quite a severe and game-changing penalty (See penalties section X.X.X).

The reason for the fact that the ball was often classified as being in the Ball Holding position before grab is executed was concerns for reliability. To grab the ball the dog needs to walk into the ball at a reasonable speed while looking at the ball. As a result the head will tilt down following the ball under it's chin. The strategy will know when the ball is in the right position to be grabbed through testing whether the head tilt angle is less than a specified value, such as -55 degrees. Decreasing this threshold will increase the reliability of the grab and whatever move is executed after the grab because the ball will be further underneath the chest in the grab.

Because of this aspect the strategy would need to be changed when incorporated into the final game strategy but in its current state the SS provided great information and input into the development of the final game strategy.

7.6.3 Improvements

Using the initial observations and results of the strategy playing the game there were a few improvements to the original strategy. A Back-Off condition was incorporated in the strategy. Improvements were not implemented extensivley though. This was the case because of the purpose for which the SS was developed, and thus perfection of the SS was not the ultimate goal.

Back-Off Condition

To counteract the dogs working against themselves it was decided to introduce a backoff condition. The backoff condition was added into step 2 in the Strategy description at the start of this report. The ammended strategy is shown below.

Strategy

1. **I**F lost the ball THEN findball gravitating towards the home position
2. **I**F found the ball **T**HEN Approach the ball
   - **I**F we see a teammate as well as the ball and the teammate is close **T**HEN back off and turn towards the target goal
3. **I**F close enough to grab the ball **T**HEN Grab the ball
4. Pan the head from left to right, or right to left looking for a goal, whether target goal or own goal
5. IF we see target goal **T**HEN check angle,
   - **I**F distance small **T**HEN chest push
   - **I**F angle small **T**HEN align and front kick
   - **I**F angle large **T**HEN do a turn kick
6. **I**F we see our own goal **T**HEN do a turn kick in the appropriate direction
7. **I**F we see no goals **T**HEN use world model information, and turn kick towards target goal

This BackOff Condition provides a solution to the problem of a dog attacking a teammate when he has the ball. The trigger however will only activate when the dog see's both the ball AND a teammate in view. If a teammate is not actually recognised then the backoff will not be triggered.

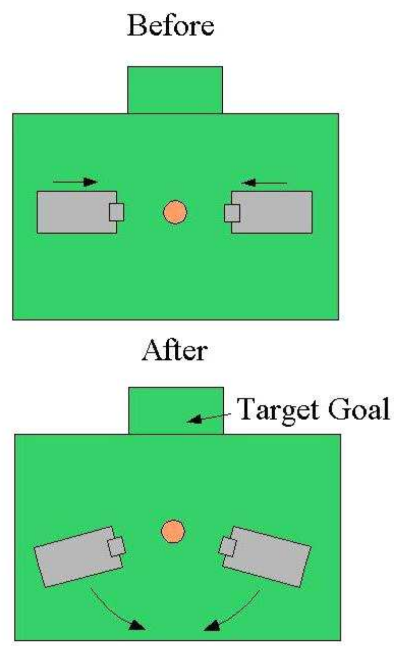


Figure 7.13: The Dead-Lock Breaking manoeuvre.

The BackOff Condition is a useful addition to the strategy in preventing team dogs from crowding the ball and each other. However the condition will only provide these advantages when the series of actions performed when the dog is in BackOff mode is well thought out and correct. The BackOff Action is just as important as the BackOff Condition.

The BackOff Action employed in SS was to move in such a way that the dog circles the ball at a distance in the direction which will cause the dog to align itself with the goal. This BackOff Action takes care of the situations in which the dogs approach from opposite sides of the ball. Such a case requires a dead-lock breaking manoeuvre otherwise the dogs will never get to the ball. The manoeuvre is shown in Figure 7.13.

7.6.4 Conclusion

The development and experimentation of SS brought great input into the final game strategy that was produced. There were a number of key input areas. Firstly, it showed the importance of attacking the ball and controlling the ball in game and this facet was counted much more highly in the final strategy. Secondly, the SS brought an increase in use of the Vision Model data. This in turn reduced the dependence of the strategy on the World Model data. These two changes carried much of the responsiveness of the SS into the final strategy.

Chapter 8

Goal Keeper Strategy

Section author: James Wong

8.1 Chapter Overview

The goal keeper's role is to stop any ball from entering the goal. Whilst this may sound easy, the larger field this year has really had a negative impact on the goal keeper's localisation — especially since the goal keeper is at one end of the field. Various improvements and modifications were made to the goal keeper's behaviour as well as infrastructure to cater for this. These improvements and modifications are discussed in the following sections.

8.2 General Strategy

8.2.1 Section Overview

The general strategy of the goalie firstly is to prevent the ball from being kicked into the goal (as opposed to being dribbled into the goal). If the ball is being kicked at the goal by an opponent (or anyone for that matter!), the goal keeper should try its best to block the ball and prevent the ball from entering into the goal. If the ball however is being dribbled to the goal (or is travelling slowly to the goal), then the goal keeper should attack the ball and prevent the ball from getting any closer to the goal. The purpose of the goal keeper however does not consist only of preventing the ball from entering the goal. The goal keeper should ensure that the ball is pushed up the field to allow its teammates to score a goal. The three strategy issues that a goal keeper face are:

1. When to attack the ball?
2. When to return to goal?

3. When to trigger a block?

8.2.2 When to attack a ball?

The issue of "when to attack a ball" inherently means that the question of "when to stay in the goal?" needs to be answered. If the attack was made too early, then the goal keeper risks leaving the goal wide open for the opponents to pass the ball to another opponent and score. If the attack was made too late, the ball will be much closer to the goal and increases the likelihood of the opponent scoring. Furthermore, if the attack was made too late, the ball might be kicked behind the penalty line, in which case, the other teammates would be helpless in getting the ball up-field. There has to be some sort of balance in aggressiveness. The simple answer to this question is "attack when the ball is close enough". However, defining when the ball is close enough is made significantly more difficult by the fact that the localisation information is not always accurate (and may be highly inaccurate if the goal keeper has not seen any beacons for quite a while). The attack condition has to be reasonable to minimise the risk of being too aggressive or too passive.

The condition used for triggering the attack is:

- goal keeper has visual sight of the ball AND
  - ball is within 60cm of the center of the goal OR
  - ball is behind the penalty line OR
  - ball is within 25cm of the goalie

For any of the decisions that the goal keeper makes, it first has to make sure that it visually 'sees' the ball. The definition used for "ball is close to goal" in this case is that the ball is within 60cm of the goal. The condition that the goal keeper should attack when the ball is behind the penalty line is motivated by the fact that the goal keeper should push the ball up-field. The condition that the goal keeper should attack when the ball is within 25cm of it serves as a fall-back (or safety mechanism) for triggering an attack even when the localisation is really bad (distance to the ball is independent of the reliability of the localisation information).

8.2.3 When to return to goal?

The question of "when to return to goal" is made complicated again, by the fact that the localisation is not 100% reliable. The condition to "return to goal" would have to be lenient enough to allow for the fact that the localisation isn't accurate. If the goal keeper returns to goal too early, it risks the opponent scoring a goal. If the condition to return to goal is such that the goal keeper is overly aggressive, then the goal keeper would be drawn too far out of the goal and leaves the goal wide open. The situation where the goal keeper is drawn to the half-way line is undesirable. The inverse of the attack condition to return to goal cannot be used because this will lead to a jittering behaviour. This is again caused by the fact that the localisation (and even the distance measurement) is not 100% stable and the goal keeper will keep jumping from attacking the ball and returning to goal very rapidly. The condition used for when to return to goal is:

1. ball is not ‘lost’ AND
2. ball is ‘far’ enough from the goal AND
3. ball is not behind the penalty line.

The definition of ‘far’ is not the negative of “ball is close to goal” (in the attack condition). The definition of ’far’ is actually approximately twice that of “ball is close to goal”. This is to ensure that there is no hesitancy in terms of making a decision of whether to attack or return to goal.

8.2.4 When to trigger a block?

Something that was not done last year but is done this year is to trigger a block when it seems appropriate. The goal keeper should trigger a block when there is a ’shot’ made at the goal. i.e. the ball is travelling fast. As would be expected, the goal keeper should trigger the blocking action earlier if the ball is “very fast” than it would when the ball is just “fast”. The goal keeper should not block if the ball is “not fast”. If the blocking condition is too trigger-happy, then the goal keeper will be (unnecessarily) slower at getting to the ball. Even if there is a shot being made at the goal, if the ball is out of reach of the goal keeper, then it would not make an attempt at a block. Since triggering the block would be useless because the ball is out of reach of the goal keeper, it is better for the goal keeper to step sideways towards the ball, because there is a good chance that even though the ball is fast and is travelling somewhat towards the goal keeper, the ball will miss the goal.

The goal keeper in this case will step sideways and attack the ball.

8.3 Roles

8.3.1 Section Overview

There are three roles that the goal keeper can take: Aggressor, Assistant and Head Home.

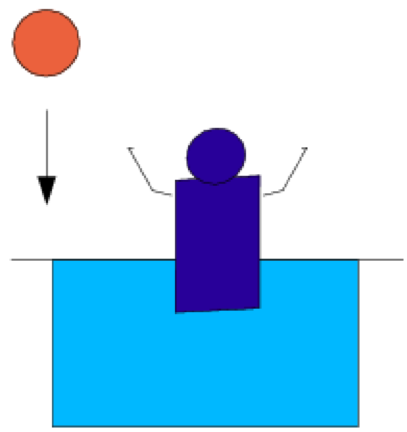


Figure 8.1: Ball out of reach

8.3.2 Aggressor

The goal keeper uses the forward's attack code to attack the ball (unless the goal keeper is blocking the ball). There is a difference between how the forward approaches the ball and how the goal keeper approaches the ball. The goal keeper has to be careful not to approach the ball in a way that may lead to the goal keeper accidentally tapping the ball in — which is more likely to happen with a goal keeper than a forward because the goal keeper is mainly near the goal. The goal keeper will get behind the ball if the ball is closer to the goal than the goal keeper or the goal keeper can see its own goal. There are a few scenarios in which the normal "get behind ball" strategy that the forward uses will not work with a goal keeper. One such scenario is when the goal keeper is at the sides of the field (after chasing the ball for instance) and the opponent somehow manages to get the ball near the goal again as the goal keeper is returning to its normal goal keeper position.

In this case the goal keeper will see the ball and will "attack" the ball. If the goal keeper circles around the ball according to the forward's code (i.e. clockwise) then the goal keeper will keep hitting the edge of the field and eventually tap the ball into the goal. What the goal keeper needs to do is circle the other direction (counter-clockwise). By circling around the ball counter-clockwise, the robot will place itself between the ball and the goal. This of course means that the goal keeper will take longer to attack the ball but the risk of the ball being tapped into the goal far outweighs the time taken for the goal keeper to circle around the ball in a way that will prevent the goal keeper from hitting the edge of the field.

```
if (ball is to the left of the robot)
{
  if (ball is left side of the field)
```

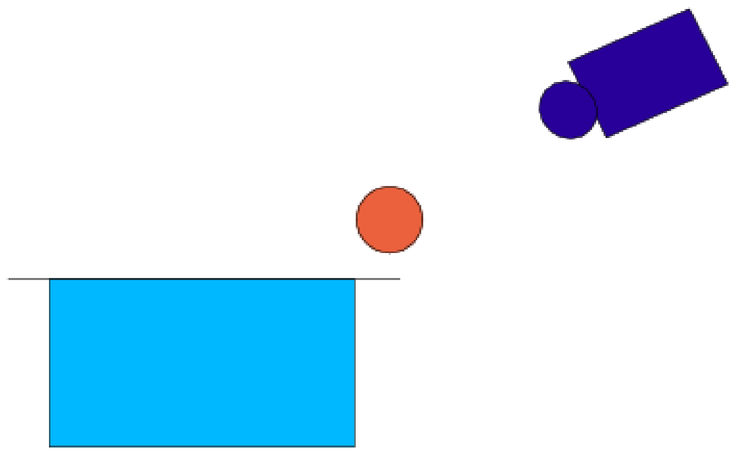


Figure 8.2: Ball is closer to the goal

```
  {
    turnDir = CLOCKWISE;
  }
  // when ball is closer to goal and the ball is on the right side of the field
  else
  {
    turnDir = ANTICLOCKWISE;
  }
}

else
{
  if (ball is to the right of the robot)
  {
    turnDir = ANTICLOCKWISE;
  }
  // when ball is closer to goal and the ball is on the left side of the field
  else
  {
    turnDir = CLOCKWISE;
  }
}
```

8.3.3 Assistant

The “Assistant” role is basically the role that the goal keeper is in when it is not attacking or returning to goal. It really should be the “Defending” role. In the “Assistant” role, when the goal keeper can see the ball it will continually try to face towards the ball and position itself in the goal in a way that will most effectively block any ball shot at the goal.

8.3.4 Head Home

The "Head Home" role specifies that the goal keeper should head back to its usual defending position (just in front of goal). This is usually called after the ball has been cleared or the ball has been lost.
Ball has been cleared condition:

```
if (lostBall <= 6 && goal2ball.d > 115 && tempBall.y > 50)
{
    role = HEAD_HOME;
}
```

Ball has been lost:

```
if (lostBall >= 12 && !utilStationaryFindBall())
    role = HEAD_HOME;
```

In either case, the goal keeper needs to return to its defending position quickly. To get back into the defending position, the goal keeper has two options: back in or return to goal.

Back In

"Back In" means that the goal keeper keeps its eye on the ball while backing into the goal. The goal keeper will back into the goal area if it is confident that it is close to the goal area. The following code determines this:

```
if (rvar < get95CF(65) && defend_target.d<50 && prevRole!=ASSISTANT)
{
    backingIn = true;
}
```

So if the variance is small enough and the goal keeper is within 50cm of the defend position, it should back into the goal. There is a further case where it checks that its previous role was not Assistant. This is to cater for the case where the robot (whilst in the "Assistant" role) is picked up and placed in the center of the field. In this case, it is best that the goal keeper doesn't back into the goal — this takes too long. The goal keeper should turn around and get back to its defending position as quickly as possible.

The goal keeper stops backing into the goal when it's variance is too high or when it determines that it is back at "base position":

```
bool MY_PLAYER::utilAtBasePosition()
{
    return backingIn && (defend_target.d < 10);
}
```

Return to Goal

"Return to Goal" means that the goal keeper turns around (facing the goal) and head back straight towards the goal. This method is much faster than backing in (as the robot's forward movement is much faster than the backwards movement). When first called, the goal keeper will turn around in a specified direction (that minimises the chance that the goal keeper will tap the ball into the goal) until it first sees it's own goal. Once it sees it's own goal, the goal keeper's head will track the goal while it moves towards the center of the visual goal blob. The goal keeper will stop heading towards the goal when the visual goal height is greater than 130 pixels (determined through empirical testing). Once the visual goal height is greater than 130 pixels the goal keeper switches to the "Assistant" role.

8.4 Goal Keeper Localisation

8.4.1 Section Overview

The goal keeper's two beacon localisation differs to the localisation that the forwards use. The forwards use the robot's distance to the beacons to determine its localisation, the goal keeper is in such a position that using the distance to the beacon is unreliable (and also unstable). The goal keeper is at the extreme end of the field and so the difference of even one pixel of a beacon results in an error that is quite significant. The goal keeper uses the angle between two beacons to determine the localisation of the robot. Angles appear to be more reliable than distances. To use the goal keeper's localisation, a line needs to be added to GPS.h:

```
#define GOALIE
```

8.4.2 Angle Localisation

To determine the localisation of the robot, the center and radius of the circle that passes through the two beacons and the robot is first calculated. The equation for the circle that passes through the two left beacons are as follows (calculating the circle that passes the right two beacons is similar although with

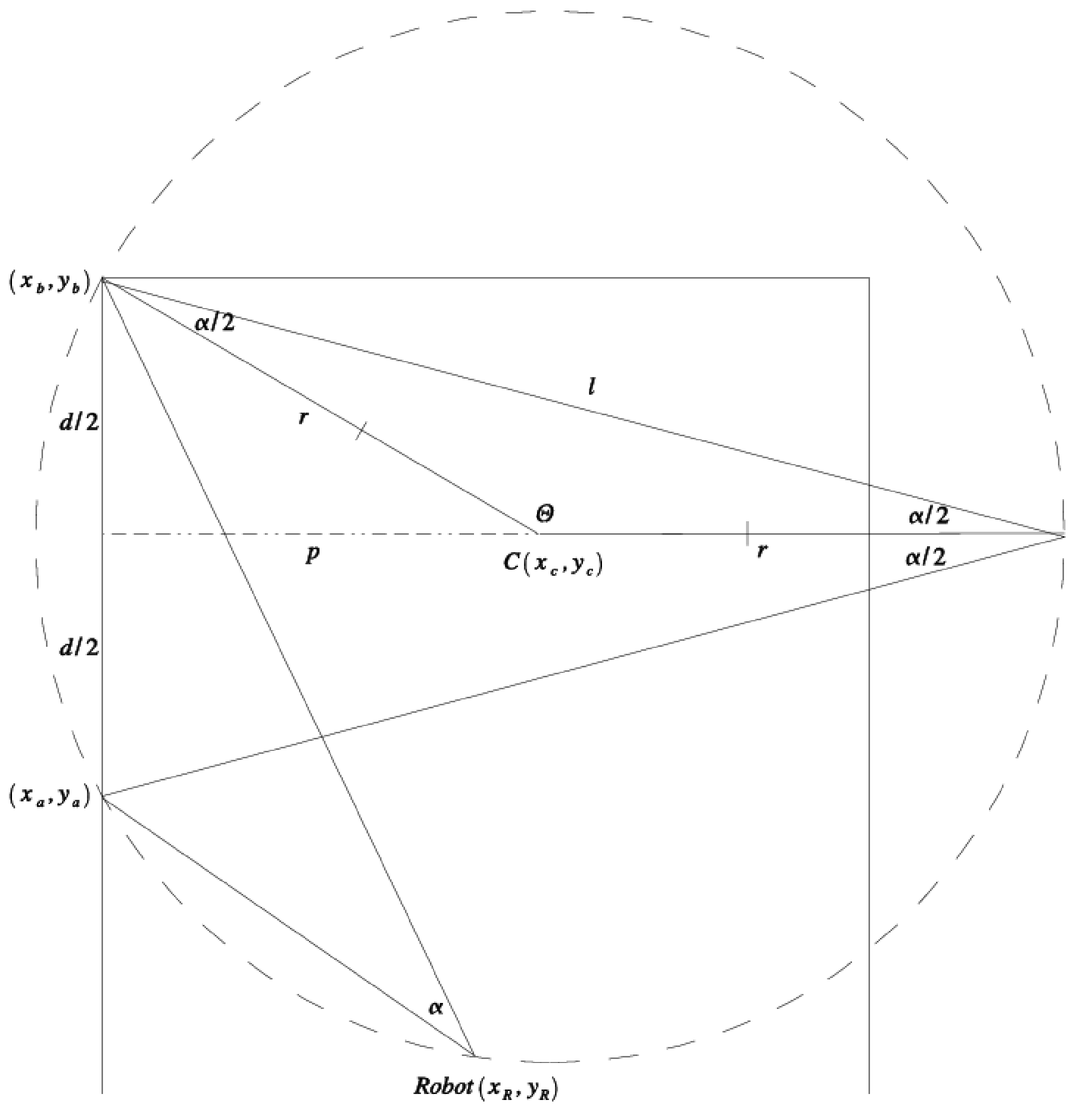


Figure 8.3: Localisation using angles

some modifications):

$$
\begin{aligned}
\sin(\frac{\alpha}{2}) &= \frac{(\frac{d}{2})}{l} \\
l &= \frac{(\frac{d}{2})}{\sin(\frac{\alpha}{2})} \\
r &= \frac{l}{\sqrt{2(1-\cos\theta)}} \\
p &= \sqrt{r^2 - (\frac{d}{2})^2} \\
x_c &= p \\
y_c &= (y_a + y_b)/2
\end{aligned}
$$

After the equation of the circle has been determined, the next step is to draw a line from the robot's approximate position to the center of the circle and determine the closest point where the line intersects the circle.The point where the line intersects the circle is where the robot is determined to be. This method is quite accurate however it requires the approximate position of the robot to be fairly reasonable. The code actually uses the last position of the robot as the approximate position. This is a fair approximation because the position of the robot is unlikely to have changed much within 1/25 of a second. However, it is a problem when the approximation is significantly wrong in the first place. There is a solution to this problem (refer to "Pan around vision ball").

8.5 Skills

8.5.1 Section Overview

8.5.2 Find Ball

The "Find Ball" routine tries to cover the area around it, whilst preventing a goal. Depending on whether the goal keeper thinks it's at the right side of the goal or the left side, it will spin in the direction that will prevent goals from being accidentally scored by the ball being tapped in by the spinning action. The code for determining which direction the goal keeper should turn is:

```
turnDir = (rx > FIELD_WIDTH/2) ? ANTICLOCKWISE : CLOCKWISE;
```

From this code, when the goal keeper is on the right side of the field, it will spin in an anti-clockwise direction. Conversely, if the goal keeper is on the left side of the field, it will spin in a clockwise direction. The direction of the spin was used in the UNSW RoboCup 2001 team code.

The find ball routine actually won't keep spinning around and around. It'll

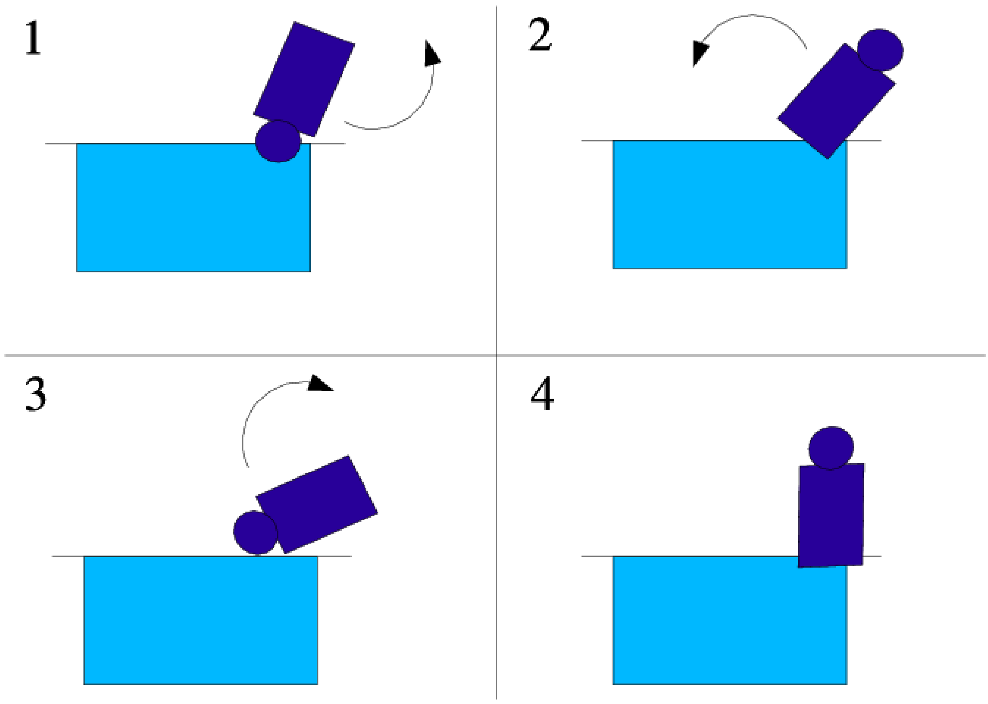


Figure 8.4: Find ball routine

sweep around the surrounding area once and then face the front and continue trying to search for the ball. From the diagram above, the goal keeper spins around in the specified direction (determined in the code above) until the goal keeper has seen the front once. i.e. robot's facing (the variable 'rh') is close to 90 degrees (within 15 degrees of this facing). After it has seen the front once, it will continue spinning in the specified direction until it spins a little past the x-axis at which case the "find ball spin" is deemed to be complete and the goal keeper spins back to face the front. All throughout the spin, the goal keeper is executing the searching head movement (i.e. keep rotating its head).

8.5.3 Block Ball

The blocking condition relies heavily on the "fastBall" variable and whether the ball is "in reach" in determining whether to block and when to trigger the block.

Fast Ball

Initially there were only two states that the ball could be in: FAST and NOT_FAST. However, after extensive testing it became obvious that in some cases, the block condition was triggered a bit late — hence the introduction of a third state, VERY_FAST. To actually work out whether a ball is fast, the displacement of the ball is sampled over 8 frames — this number appears to be enough to get a consistent value. Once the displacement of the ball is collected over 8 frames, this displacement is divided by 8 (the number of frames over which the displacement was collected) to give the velocity of the ball. Depending on the velocity of the ball the "fastBall" state is determined. If the velocity of the ball is greater than 2.0, then the ball is deemed to be "FAST". If the velocity is over 3.0 then the ball is deemed to be "VERY_FAST". If the velocity of the ball falls below 1.5, then the ball's state becomes "NOT_FAST". If the velocity is between 1.5 and 2.0 then the ball's state doesn't change. This "buffer" region prevents the goal keeper from fluctuating between behaviours because the ball's state will be more stable. The code is as follows:

```
if (ball_speed > 3.0)
{
    fastBall = VERY_FAST;
}
else if (ball_speed > 2.0)
{
    fastBall = FAST;
}
else if (ball_speed < 1.5)
{
    fastBall = NOT_FAST;
}
```

Within Reach

"Out of reach" means that even if the goal keeper is in the blocking stance, the goal keeper would not be able to "reach" the ball. To determine whether a ball is out of reach, the visual ball is used. From the visual ball, the value of the "lateral offset" can be determined. Like the fast ball value, the "out of reach" value was determined empirically — 15 works quite well. So to make sure that the ball is within reach there would be the following condition:

```
ABS(latOffset)<=15
```

Blocking Condition

The blocking condition itself is simple enough:

```
if (lostBall>=6)
{
    return false;
}
```

```
else if (fastBall==VERY_FAST && tempBall.d<45 && ABS(latOffset)<=15)
{
    return true;
}
else if (fastBall==FAST && tempBall.d<35 && ABS(latOffset)<=15)
{
    return true;
}
```

Again the distance at which the block is triggered is determined empirically.

8.5.4 Pan around vision ball

Due to the nature of the two beacon localisation using angles, the localisation of the goal keeper significantly improves when the goal keeper is scanning back and forth between two sets of beacons. When the ball isn't very close to the goal the goal keeper will continue panning around the vision ball which allows the goal keeper to:

1. maintain visual contact with the ball
2. refine its localisation

Once the ball gets close to the goal, the goal keeper no longer pans around the visual ball to avoid the risk of the goal keeper losing sight of the ball at a crucial stage.

8.6 Future Improvements

8.6.1 Section Overview

From the competition this year there are clearly quite a few improvements that needs to be made to the current goal keeper. The following are some of the improvements that may be made to the goal keeper next year.

8.6.2 Better Clearing of Ball

Once the goal keeper has control of the ball. The goal keeper should try to kick the ball at the gap upfield. Currently the goal keeper will just try to kick the ball upfield without considering whether there is another robot in front of it. So even if there is an opponent right in front of the goal keeper, the goal keeper will continue to execute its kick. Sometimes this results in the ball bouncing off the opponent and into the goal. Obviously the goal keeper would have to quickly make a decision about where to kick the ball and not waste any time.

8.6.3 Avoid Getting Stuck

When the goal keeper tries to attack a ball that is at the side, it usually gets stuck at the edge of the goal. This is sometimes desirable when the ball is close as it prevents the opponent from pushing the ball into the goal. However, when there are no opponents close by, the goal keeper really should be getting out of the goal (instead of getting stuck at the edge of the goal) and trying to clear the ball. This may not be so much of a problem (see "Less Aggressive Goal Keeper") if the proposal to make the penalty area a box around the goal is carried out.

8.6.4 Avoid Other Robots

One of the most common scenario is where the goal keeper attacks a ball and is away from the goal and as it is trying to return back to goal, it gets stuck with another robot that is along its path. In this case, it would be nice if the robot tries to avoid the other robot by strafing sideways until it is clear to continue returning back to goal. There will be problem with this strategy if the other robot is moving as well so the robot shouldn't strafe for an infinite period of time.

8.6.5 Less Aggressive Goal Keeper

The proposal to make the penalty area a box around the goal (if carried out) would mean that the goal keeper can afford to be much more conservative and stay around the goal area more. A decision was made this year to make the goal keeper attack the ball when the ball is behind the penalty line because none of its team mates can attack the ball (see "Illegal Defender" rule). This meant that in quite a few instances the goal keeper would stray out of the goal area whilst there are multiple attackers and in the midst of attacking the ball, the ball pops out and the other attackers would get the ball and kick it in. A more conservative goal keeper would allow the goal keeper to defend its goal better.

Chapter 9

Challenges

9.1 Challenge 1 Shape Recognition Challenge

The summary of the challenge as from the Open-R web site is as follows:

> *The Challenger must recognise 3 objects (2-D shape) out of 5 objects defined by Sony in the field and visit each object. In front of each object ("wag-the-tail-area"), the robot must wag its tail and play WAV file of the object 3 times. Any order for visiting is permitted.*

Figure 9.1 shows a diagram of what is to be performed.

The ultimate object of this challenge is:

> *To evaluate pattern recognition ability.*
> *In the future, illumination invariant object will be used instead of the colour poles for the localisation. Please not that the object used in this challenge this year is just a proposal. Base on this years' competition, we should discuss what kind of pattern will be suitable in the future.*

The aim of this challenge is to start to move away from the usage of colour coded objects around the field for localisation. Reasons for the changes are because soccer in real life does not depend on having colour markers around the playing field and relies on players remembering the features around the playing field. The usage of patterns around the field is only a step forward in removing the dependence of colour for the robots in determining its position, since the processing of colour is relatively simple in comparison to patterns.

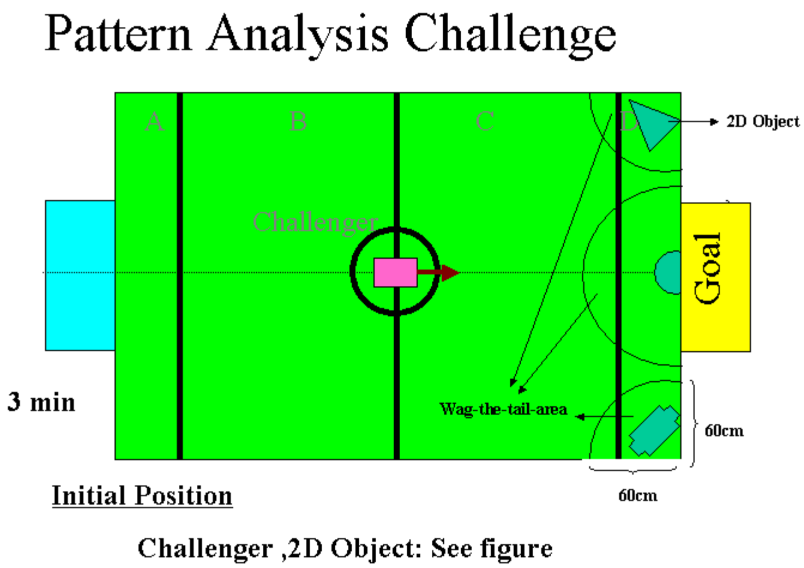


Figure 9.1: Pattern analysis challenge

9.1.1 Overview of rules and restrictions

Definitions

- **wag-the-tail-area**
  - the area within 600mm diameter of each object
  - robotic dog must have at least 3 feet in diameter when playing sound file, else not deemed to be in wag-the-tail-area
- **2D Shape**
  - black and white checkered patterns;
  - should not be segment the object by colour;
  - be placed in 3 fixed locations; front of goal and in both; corners on one side of the playing field;
  - the exact shape is not announce before competition;
  - shapes may be rotated, varied in size, but fixed in frequency (each square in patter is 20mm in size)

Rules

Initial Settings See Figure 9.1 for initial setup.

- 3 objects are put at the yellow goal, and both corner
- Robot is put on the center of the center circle faced to the yellow goal
- start challenge with whistle
- push switch in order to start program. Any leg motion before switch push is not permitted (head motion for localisation is allowed).

Step 1

- Robot goes to 1st object and wags its tail (to notify referee) and play a sound file of the object name 3 times inside the wag-the-tail-area.
- If the robot wags its tail out of the wag-the-tail-area, challenge is over
- If the robot plays the sound out of the wag-the-tail-area, challenge is over

Step 2

- Robot goes to 2nd object and wags its tail and plays sound file of the object name 3 times inside the wag-the-tail-area
- If the robot wags its tail out of the wag-the-tail-area, challenge is over
- If the robot plays the sound out of the wag-the-tail-area, challenge is over

Step 3

- Robot goes to 3rd object and wags its tail and plays sound file of the object name 3 times inside the wag-the-tail-area
- If the robot wags its tail out of the wag-the-tail-area, challenge is over
- If the robot plays the sound out of the wag-the-tail-area, challenge is over

Shapes

The shapes to be used in the robocup pattern analysis challenge are the square shape (Figure 9.2), rectangle shape (Figure 9.3), triangle shape (Figure 9.4), T shape (Figure 9.5) and the L shape (Figure 9.6). During the competition, only three of the outlined shapes will be tested. Each shape will be filled by a black and white checkered patter, each of 20mm x 20mm in size.

Each shape in the challenge may additionally be of variable size, while retaining the size of each checker pattern.

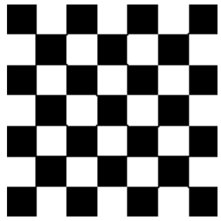


Figure 9.2: Square shape

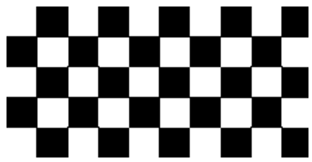


Figure 9.3: Rectangle shape

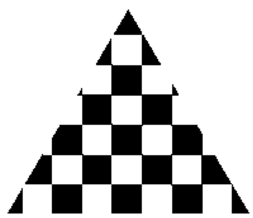


Figure 9.4: Triangle shape

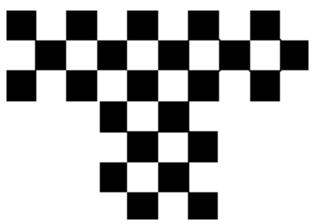


Figure 9.5: T Shape

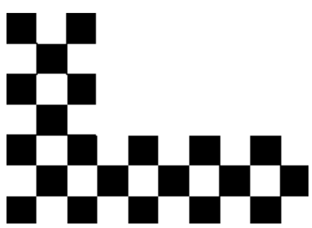


Figure 9.6: L Shape

9.2 Challenge 1 Approach One

Section author: Albert Chang

While there was a shape recognition challenge in the Robocup 2001, this years challenge extends on last years challenge by further extending it. Instead of having flat colours which where distinct in some respects to the robots environment, the shapes rather were filled with a pattern. It may not be so obvious in that the transition from the recogition of flat coloured shapes to pattern-filled shapes was a significant step. The complexity arises due to the following:

- Detection of flat colours is a trivial task. Often accomplished very well through thresholding or any other similar methods that attempts to map the shape to a particular signature.
- Regions containing various different patterns difficult to distinguish with a low time complexity algorithm (a fast algorithm).
- Possibility of distortion of pattern makes recognition difficult. Such distortion could include perspective distortion or difficiencies in camera.
- Most proven methods often developed under lab conditions where noise in images is minimal.

Due to the reasons as outlined above, various techniques were examined and the following is an outline of some that were explored.

9.2.1 Literature Survey

Image Processing Handbook

John C. Russ, Image Processing Handbook, 2nd Edition, CRC Press Inc, 1994 [pp283-346].

[Rus94]On reading this book, some interesting information was obtained about viewing images in the spacial domain, and frequency domain. What was of interest was viewing images from the frequency domain.

Fourier's thereom states that it is possible to form any one-dimension f(x) as a sumation of a series of sine and cosine terms of increasing frequency. The Fourier transfom of the function f(x) is written F(u) as describes the amount of each frequency term that must be added to make f(x).

Similarly, a Fourier transform may be computed on an image, by computing a Fourier transform on the horizontal and vertical axises of the image, to produce what is often called the frequency domain of the image. From this transform, measurements are much more easier to perform, especially when using a computational device such as a computer. Measurements may be performed just by analysing the period of repetition, in the frequencies in the transform.

Additionally, the orientation of features may be easily determined much similarly to measuring the size of features. The orientation of the repeating patterns follow that of the original image.

An Algorithm for the machine calculation of complex Fourier sets [JC65]

This paper presents us with an effecient method of calculating Fourier sets, most suited for execution on a computation device such as a computer. What is presented here is what is known as the Fast Fourier Transform, otherwise known as FFT. This algorithm was the choosen algorithm used to produce Fourier sets.

Image Pattern Recognition [spi87]

This was a set of papers presented at a SPIE conference in California, presenting some interesting findings in regards to Image Pattern recogition. The section of most importance to my work was in regards to a Review of Machine Vision and Architectures [spi87].

This paper presented an overview of how an image processing systems may be broken down, and implemented by presenting a framework by which an architecture could be implemented for image processing. The steps of image processing is generalised into six phases:

1. Enhancement
2. Feature Extraction
3. Morphological Processing
4. Symbol Processing
5. Mesuration
6. Application-dependent processing

Enhancement

This phase is mainly in dealing with:

- Compensation for lighting and camera deficincies.
- Reduce camera noise.
- Sharpen image detail.

This phases indeed is regarded as an important step in allowing succesive steps from being processes effectively. However, due to the conditions that the challenge is operated under, they were of little concern, and hence was a phase skipped altogether. The operation of the robots were always under adequate lighting, that was always uniform and intensity, and intensity at a suitable level. The shutter speed of the cameras in operation were always set to the slowest speed, since colour information was more important that image clarity. For this reason, noise was minimal. Most images obtained often contained good clarity, and so, enhancement was seen as process intensive tasks that wasn't required. However, as rules and condition change in the future, this phase may need introduction to ensure reliable operation.

Feature Extraction

This phase is mainly in dealing with:

- Detection of edges to isolate object boundaries.
- Recognise texture variations.

This was an essential step, and was one that was in the core of the task at hand. Before an object recognition could occur, features need to be identified, and isolated to increase the operation performance of following processes. For this stage, we ourselves attempted several methods to ensure a good attempt at solving the problem at hand is obtained.

Morphological Processing

Generally, this involves processes such as:

- Dilation/erosion of objects.
- Skeletonization of objects.

Morphological processing often requires operations that are peformed on an image, often performed several times before any results are evident. Due to the fact that dilation and erosion operations may need to be performed several times, makes it an unsuitable choice for real-time image processing.

Symbol Processing

Symbol processing involves:

- Segmentation of images into regions of common properties.
- Generate shape descriptors of objects.

- Generate statistical descriptors of objects.

This is an important step in any image processing application and hence is a stage that cannot be missed. For our task, it would include the isolation of shapes and the paramaterisation of the shape itself.

Mesuration

This stage is in the dealing with measurements of the symbols. This would include finding the area of the shapes, perimeter of the shapes or even finding the moments of inertia of the shapes. Many other applicable calculations may be performed here.

Application-dependent processing

After the previous steps have been followed, what is left is the collating of results. This may include the use of artificial intellignece in making decisions about objects and other additional calculations may be performed in the process.

Digital Image Processing Techniques [Eks84]

This piece of literation provided an insight into texture analysis. While the task that we were to perform requires us to identify shapes, filled with a checkered pattern, then a step was required for us to process the checkered pattern. It was anticipated that many of the methods produced for the Robocup 2001 Shape Challenge, may have been of usage, provided that filtering of the checkered pattern was possible, hence this area was examined.

According to Ekstrom, the analysis of textures may be performed by calculating the Fourier power spectrum at various bands. Other methods also included the calculation of the second order statistics of paris of grey levels at various seperations, and the first order statistics of absolute gray level differences at various seperations. While these methods were identified, it was acknowledged that pattern analysis was not a trivial task to perform. However, combining with other knowledge of Fourier transforms from previous references, it was seen that Fourier transforms could be examined furthermore.

Other methods of pattern analysis include "global feature detection and template matching". This method is a brute force method, where patterns are compared with convulved copy of the desired pattern. If a match is found, we can start to mark areas of the pattern out, until the entire region is identified.

Computer vision for Robotic Systems: An Introduction [Fai88]

This piece of literation, provides us with a method of making decisions on the determination of what shapes features are actually are. The following steps were outlined:

- Define objects in terms of its constituent features which are the elements to be worked withn.
- Decision tree may be used to categorise objects

By identifying features, of seen objects, then we can simplify determine what shape an object represents by examining the features using a decision tree. This provides an implementation that makes decisions rapidly, and allows easy implementation.

9.2.2 Methods and Processes

The stage that most object recognition systems work is as follows.

1. Image segmentation
2. Feature Extraction
3. Symbol Mapping

While these processes and methods are much similar to those as specified by [spi87], however has been simplified to somewhat. Futher sections explain in greater detail processes that were examined.

Image segmentation

As in most image processing applications, the first stage is to identify important aspects of an image that is relevant to us. This stage is often called image segmentation. There are a variety of methods that exist that can help us to identify useful regions in an image. A discussion follows on methods that were examined and trailed in the case of shape segmentation.

Usage of colour classification

Due to the specialised properties of the shapes, it was possible to use colour classification in order to segment the image, to identify regions in the image identifying a shape. Since objects placed on the field had distinct, disparate colours, it was possible to mark out areas in images, which possibly contained useful information, such as the presence of an object.

Fourier Transformation

For the shape recognition challenge in Robocup, it was stated that all shapes would be filled with a 20x20 millimetre black and white checkered pattern.

Given that this was a very regular repeating pattern, a Fourier transformation could be used as a filter to identify the repeating pattern in the image.

The technique used to identify shapes is similar to the technique of using Fourier transformations to detect edges in images. Since the checkered pattern is composed of black and white squares, it can be said that the shape is made up of a high contrast pattern. Since high contrasting pixels in the image would produce high frequencies with the Fourier transformation.

Once we have determined the highest frequencies in the Fourier transformation, we then are able to determine the orientation of the checkered pattern and its size. By using this information, we are then able to transform the original image such that each box in the checker pattern occupies one pixel, and that rows and columns of the checker pattern are always horizontal and vertical. Since this process puts the identified shape in a standardised orientation, then a template matching operation can be applied to reveal the actual shape.

To begin with, refer to images presented in Figure 9.7 and Figure 9.8. Firstly, we have an image of a square and some Fourier transformation. In Figure 9.7, it can be seen that the high contrasting square in the image appears as repeating white blobs towards the corners of the Fourier image. The repeated blobs decrease in intensities as it moves away from the corer of the image. Now in Figure 9.8, we have an image with 5 white squares, and it's corresponding Fourier transformation. As can be seen, by repeating our original white square in the original image, produces an increase in intensity for the blobs in our Fourier transformation. We use this property to our advantage in the detection of checkered patterns in our images.

New on Figure 9.9, we translated our checker pattern away from the center of the image. The result of the translation of the pattern brings about minor changes in our Fourier transfromation, and it is evident that our method of detecting repeating patterns is still viable. The Fourier transformation proves to work indendent of the position of the pattern in the image.

The next property about the Fourier transformations that is important to us is shown in Figure 9.10 and Figure 9.11. These two figures presents us with images where each shape has been scaled down to 50% of the original shape. When we perform this transformation on the original images, then the corresponding Fourier transformation would have the repeating blobs greater distances appart. This information is used to our advantage in determining the viewed size of the checker pattern. This property allows a method to be produced that allows the checker pattern to be detected, independent of its viewed size.

The next set of images presented in Figure 9.12, shows an image, having a similar repeating pattern as that presented in Figure 9.11, but has additional noise added. The results of the Fourier transformation can be seen in the image on the right. If we were to compare this transformation with that presented to us in Figure 9.11, then there minimal difference in the repeating blobs. The repeating blobs of high intensity still follow the same pattern, while small differences do appear in the corner areas of the transformation. Since it is the repeating property of the Fourier transformation that we are interested in, then those small changes within the corner of the images are not much importance to us. It is this property that aids in limiting background noise from affecting our results.

Finally, in Figure 9.13, we are presented with an image which is rotated to some angle. Similarly, if the original shape was rotated, then the repeating blobs of the Fourier transformation is also rotated at an angle similar to that of the shape's rotation.

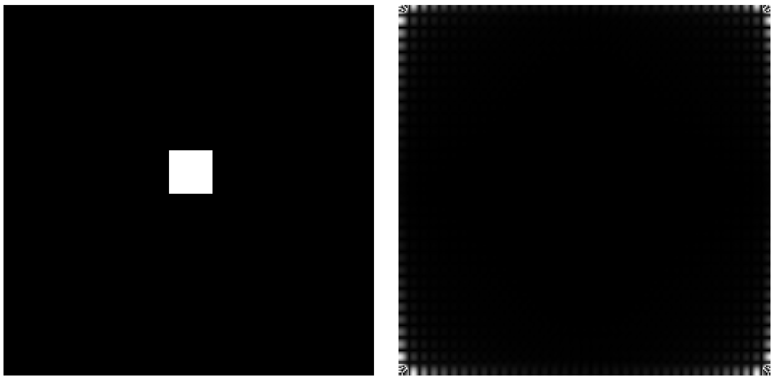


Figure 9.7: A simple square shape and it's Fourier transformation

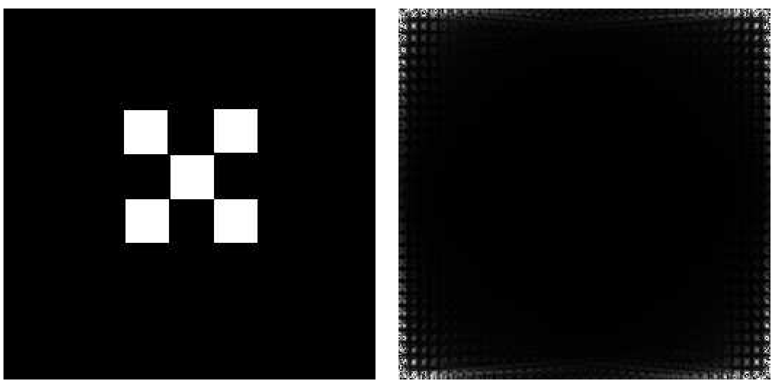


Figure 9.8: A 3x3 checker pattern and it's Fourier transformation

As can be seen, the properties that have been outlined makes Fourier transformations as a feasible solution to the outlined problem. The properties are the following:

1. Pixels of high contrast have produce more high frequencies in the Fourier transformation (white blobs on the images corresponding to the Fourier transformation).
2. The Fourier transformation is computed on the complete image, producing data that reflects the image globally. Allows this method to be independent of the location of the pattern in the image.

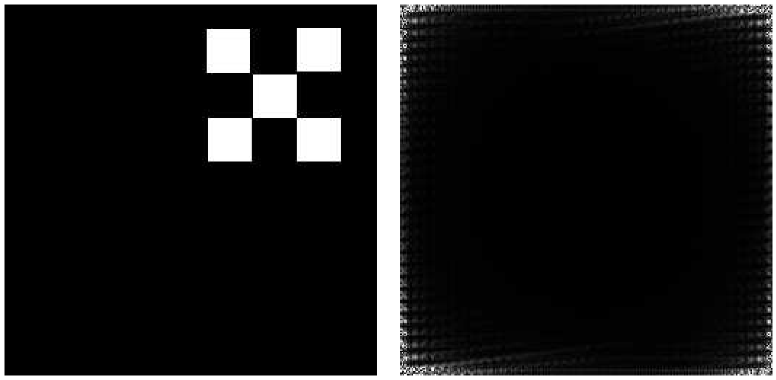


Figure 9.9: A 3x3 checker pattern translated to a different position from that of Figure 9.8 and it's Fourier transformation

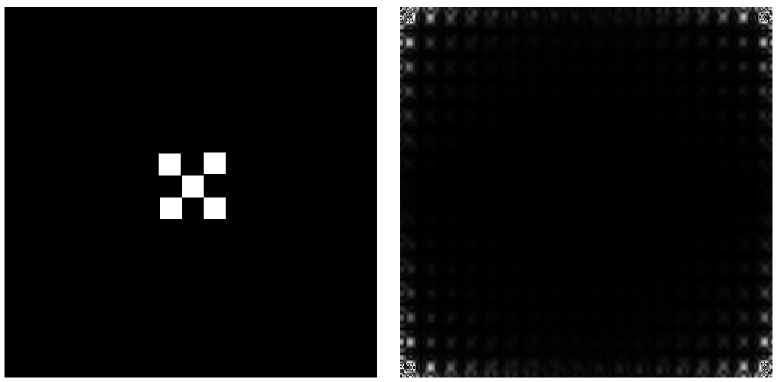


Figure 9.10: A 3x3 checker pattern after scaling to 50% of the original size and it's Fourier transformation

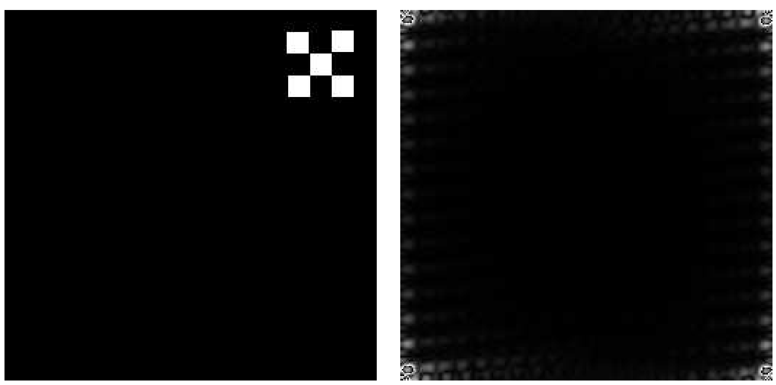


Figure 9.11: A 3x3 checker pattern after scaling to 50% of the original size and translated to the top-right of the image and it's Fourier transformation

3. The size and orientation can be extacted from the Fourier transformation, making it possible to reposition image, such that our pattern can be

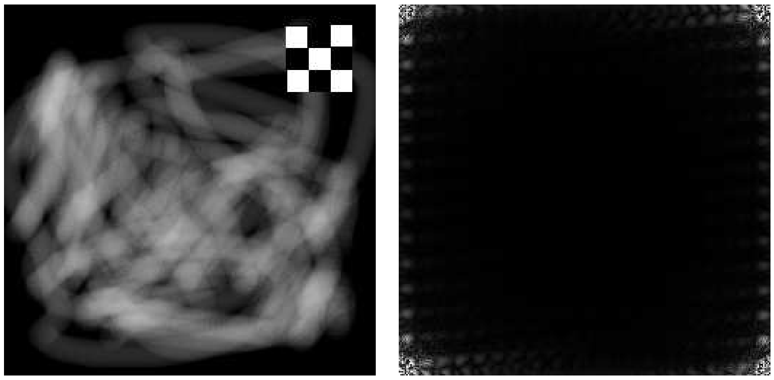


Figure 9.12: A 3x3 checker pattern after rotating 90 degrees and it's Fourier transformation

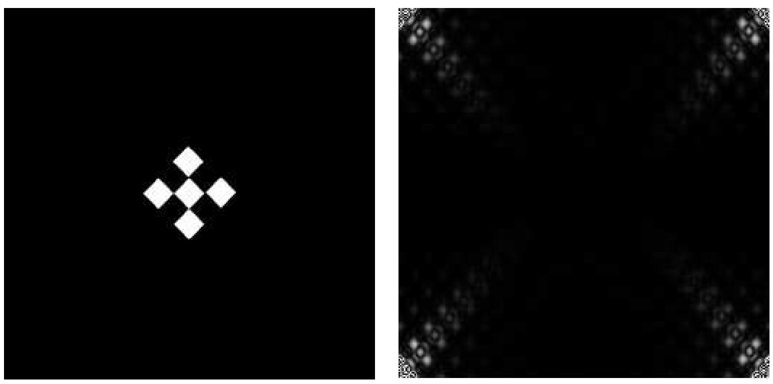


Figure 9.13: Fourier 7

checked simply by scanning rows and columns in the image.

4. Provided that the checker pattern is of high-contrast, the produced Fourier transformation is not affect significantly by background noise.

By using the Fourier transformation, to help us to extract features, and reposition features, such that the pattern can be masked out, then the Fourier transformation can be of aid to the problem at hand. The steps that may be taken is outlined in Figure 9.14.

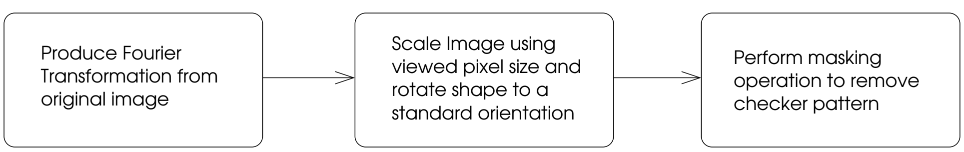


Figure 9.14: The process of shape recognition using the Fourier transformation

Figure 9.15 shows a sample session of what is performed in order to reveal a shape. The first presented image is the original image (image a). This image is taken directly from the camera. The second image is one where the perspective has been corrected (image b). The correction of the perspective is not an essential task but it improves the accuracy of the method. By minimising the effects of perspective, we get more regular square shapes. From the perspective corrected image, we calculate the Fourier transformation, using a FFT (Fast-Fourier transformation) algorithm. From the Fourier transformation, the maximal frequencies are calculated, from which we determine the orientation of the checkered shape, and its size. Using this information, we scale the perspective corrected image (image b) to produce image d. As can be seen in image d, each square on the shape corresponds to one pixel in image d. Additionally, it can be seen that the shape in image b is slightly slanted but image d corrects the slant, by using the positions of the squares to its advantage. The final stage was an additional post-processing stage to lift or remove the checkered pattern.

World Model Information

Since the vision system is one in many systems in the robots, it was possible to obtain higher-level information about the robot, such as it's position relative to the playing field. Since rules outlined that shapes would be placed at fixed locations, it was possible to determine the approximate location of objects based on robot position, robot's viewing direction and so on.

While there was a certain amount of error in the world model information, it was still of usage when we could determine that world model information was of high reliability. By having information of where the robot was, the position of the camera, and its viewing displacement relative to the field was known. Since competition rules outlined that shapes were placed in fixed location, it was possible to pre-program the robots to move to locations on the field where the shapes were placed. Next, the positioning of the camera could be placed such that it looked at the shape. This would allow the robot to perform the image segmentation for us. As can be seen in Figure 9.17, our robot knows the height that the shape will be placed at, and can itself can determine the distance of the shape, based on what it knows about the playing field and where itself is located. This information is enough to perform a rough scale to translate the patterns from its viewed size to a uniform size.

For other tasks such as the Fourier method described in Section 9.2.2, the perspective correction was useful in that it allowed the perspective correction parameters to be determined. These values were often pre-programmed as the robots were often programmed to move into fixed positions relative to the shape locations.

Other Image Pre-processing Techniques

Using some other appropriate filters, it was possible to identify regions on the images, which was where shapes occupied. Most methods under this category worked quite well, however required high levels of processing power to work effectively, and required pre-computed values often obtained by experimentation.

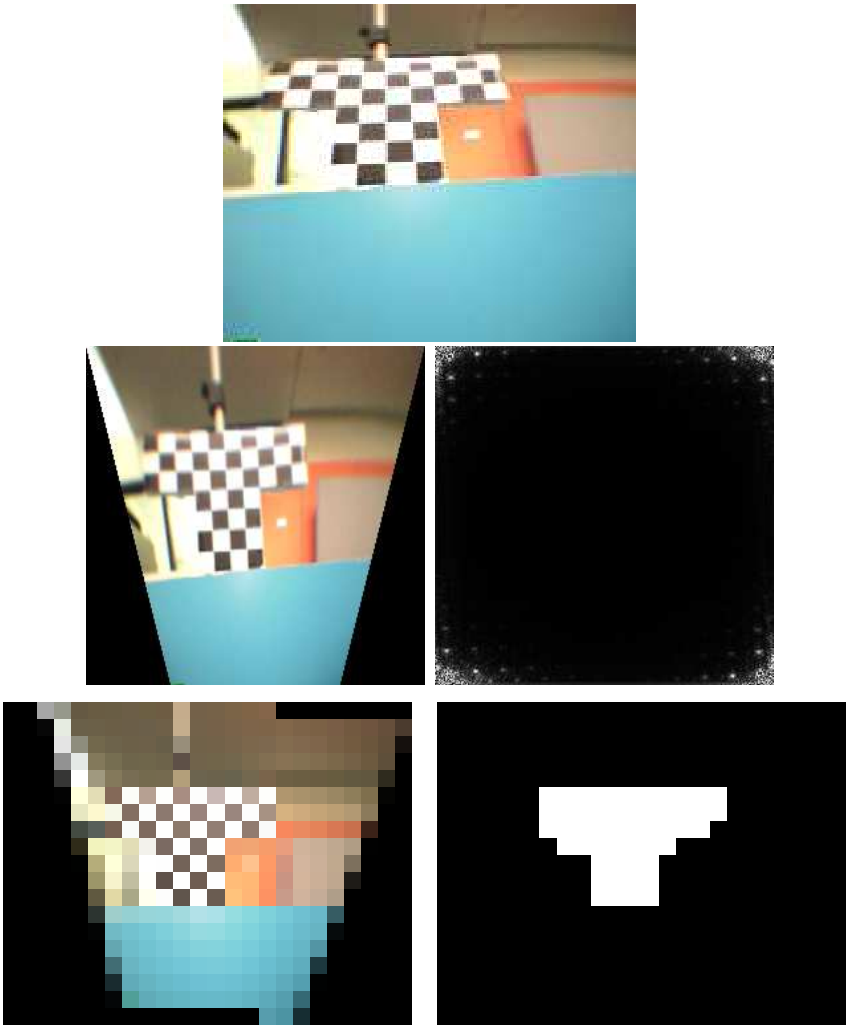


Figure 9.15: Images in order from top to bottom, left to right. a. Original image, b. Perspective corrected image, c. Fourier transformation, d. Translated and scaled image, e. Masked image

9.2.3 Feature Extraction

In most image processing applications that deal with object recognition, one important phase is feature extraction. Feature extraction is the process of identifying important areas in an image that is of important to us. In the case with Robocup, this included shapes from the shape recognition challenge and objects on the field such as the orange ball, goals, and the beacons surrounding the playing field.

If an image is viewed in its entirety, where each pixel is represented by a 24 bit value, then each pixel would be able to have over 4 billion possible states, then it is quite a difficult task to process relationships between pixels within an given

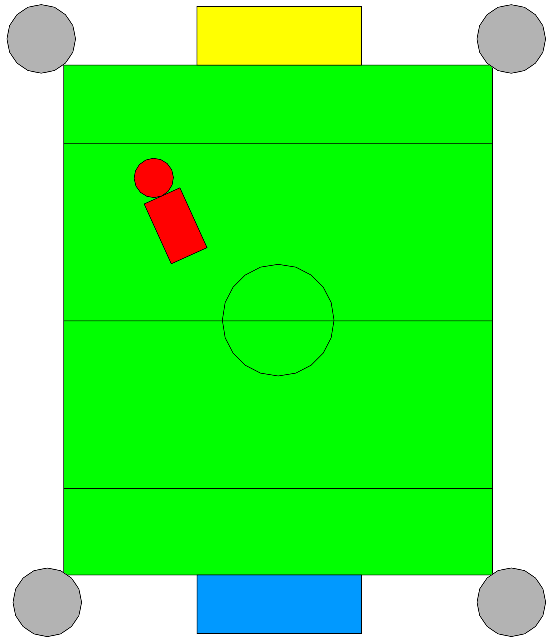


Figure 9.16: The dog knows it's position on the field

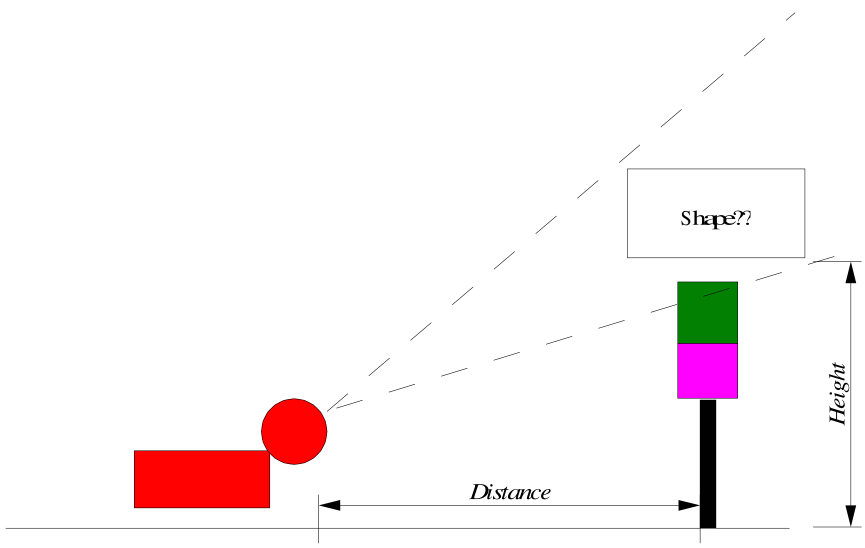


Figure 9.17: The dog can move up to the shape, and since it knows it's distance from the shape and it's height and pattern size is known, the scaling operation can be performed using information it already has.

area. Such image processing tasks can be performed quite effectively by the human vision, since humans are quite attuned to patterns and the processing of patterns.

Thresholding

This is a technique that relies on colour information of an image. Through thresholding, we map all the possible states that a pixel may have to a smaller set. By using this technique, it flattens the "colours" of objects seen by the vision system. It may be observed that high colour images taken from a colour camera often contained a level of noise. If an image was taken of a shape having a flat colour, then the pixel values making up the picture often was not uniform, having values above or below actually within the range of the noise level. This simple example illustrates immediately the problem of an attempt of identifying a coloured shape. Thresholding tries to simplify the colours of an image, eliminating the effects of noise. Other factors that affect the colour of an image would also include shadows, and lighting.

For our case, where we are trying to find black and white checkered patterns, then we can distinguish between black and white. Shown in Figure 9.18 is an example of the results. As can be seen it can simplify the image creately, making it easier to process the image. This method however, is variable to lighting, as the threshold level at 75% luminosity has been pre-tested.

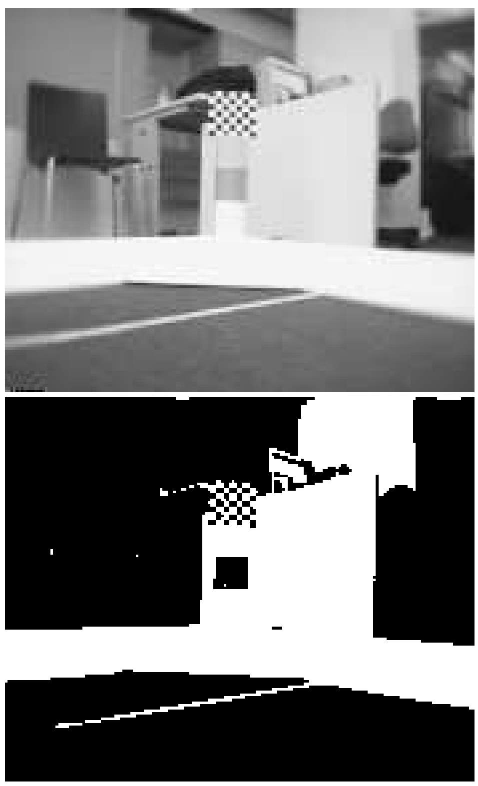


Figure 9.18: The first image is the original image, where colour channels have been removed since we are only interested in the black and white checkerd pattern only. The bottom is the thresholded image at 75% luminosity.

Colour mapping

While thresholding was a generic method of removing noise and other factors causing variations in the pixel values, it often was not specific enough. If a particular image contained many shades of red, a human would still classify the complete image as a red image, not making any distinction between the slight variations in saturation and intensity. Similarly, colour mapping attempts to simplify the possible states that particular pixels may hold, just like that of thresholding. However, the mapping is performed by using a table lookup from the actually value to a new value. By using a table lookup, we could map all possible shades of the colour red to one value. Such a colour lookup table needs to be pre-computed before usage. Since it is not a dynamic method, it can be specifically tuned to the lighting conditions of the camera operation. This method worked very well for Robocup since objects used often had flat colours that were significantly different.

Pattern removal

Since our shapes did not have a flat shape, then a method of removal pattern was required to ensure that we can find objects. Since the shapes were always filled with a black and white checkered shape, then the pattern that we aim to remove are the ones identified in Figure 9.19.

If we used the Fourier transformation method as described in Section 9.2.2, then each box in the checkered pattern occupies one pixel in the image. This is the easist case to process. Since the values of the piexls will not always be black and white and have some shade of the two, then we can determine if two pixels are high contrast by the formula:

$$contrast = \frac{min(a,b)}{max(a,b)} \tag{9.1}$$

$$ishighcontrast = contrast(a,b) > MAXCONTRAST \tag{9.2}$$

where a and b are the intensities of 2 adjancent pixels and MAXCONTRAST is a constant value that determines the minimal tolerated contrast allowed.

Following here is some C code that will return either 1, if a checker pattern is detected or 0 if no checker pattern is detected.

```
int pixelValue(int x, int y)
{
    int compareCount = 0;

    // compare pixels of high contrast
    if(contrast(pixel(x, y), pixel(x, y-1)) > MAXCONTRAST)
        compareCount++;
    if(contrast(pixel(x, y), pixel(x, y+1)) > MAXCONTRAST)
        compareCount++;
    if(contrast(pixel(x, y), pixel(x-1, y)) > MAXCONTRAST)
```

```
            compareCount++;
        if(contrast(pixel(x, y), pixel(x+1, y)) > MAXCONTRAST)
            compareCount++;

        // compare pixels of low contrast
        if(contrast(pixel(x, y), pixel(x-1, y-1)) < MAXCONTRAST)
            compareCount++;
        if(contrast(pixel(x, y), pixel(x+1, y-1)) < MAXCONTRAST)
            compareCount++;
        if(contrast(pixel(x, y), pixel(x-1, y+1)) < MAXCONTRAST)
            compareCount++;
        if(contrast(pixel(x, y), pixel(x+1, y+1)) < MAXCONTRAST)
            compareCount++;

        return compareCount == 8;
    }
```

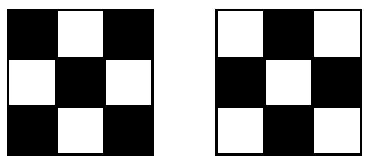


Figure 9.19: Pattern to mask

Basically, what it tries to do is whenever it encounters pixel arrangements as shown in Figure 9.19, where $pixel(x, y)$ refers to the middle pixel out of the 3x3 pixel arrangement then set it to 1, else set pixel to 0. After which processing, we should get an image similar to that of image number 5 from Figure 9.15

9.2.4 Symbol Mapping

After the pre-processing of the images, the final task at hand was to interpret what objects was being presented in the image. This is the task of symbol mapping. What is performed here, is that the identified objects in the previous steps are then mapped to a symbol, which identified what that object in the image is representing. Various methods exists that allowed this process to occur.

Template Matching

If the previous stages had revealed the shape silhouette, then one simple task that could be performed was to match the visual silhouette with a set of silhouettes stored as a library, which we call a template.

By template matching, we try to overlay our seen shape with the template in the library and analyse the differences. As shown in Figure 9.20, we are attempting to match a T shape. The first image, presents the original template image, and the second shape as the shape which we are attempting to identify. What is performed, is that we overlay our shape with the template, and examine the differences in the area. A good match is one that has minimal difference between template and shape that we are trying to match.

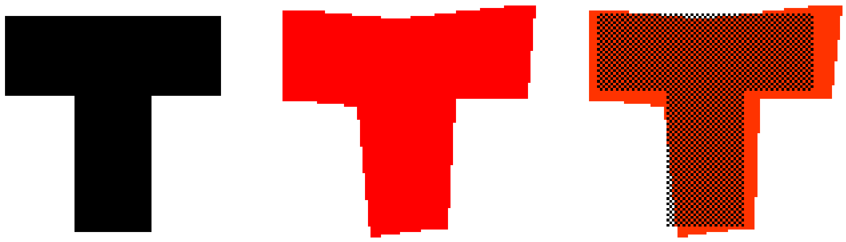


Figure 9.20: Matching a T shape template with a T shape obtained from feature extraction. Note how the shape from feature extraction is not a perfectly regular shape

Now, looking at Figure 9.21 we are trying to match a rectangle with the T shape again. This time, it can be seen that there is more difference in areas, where the ends of the top of the T do no match with the rectangle shape.

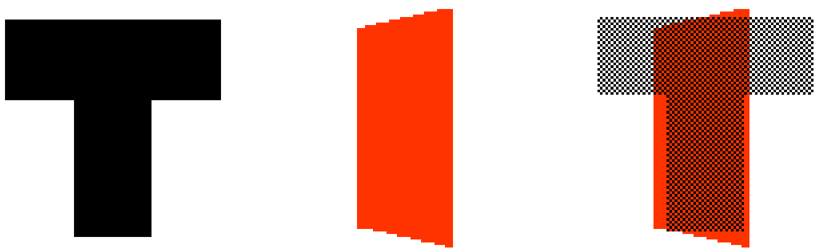


Figure 9.21: Matching a T shape template with a rectangle shape obtained from feature extraction

An obtained object would need to be compared to each template in library and the object having smallest difference is choosen as the shape. While this method is quick, it does exhibit some problems. The main problem is that it is not size and orientation invariant, a main condition in the challenge. To make this method size and orientation invariant would additional processing, making it a costly operation. Such a possible implemention would try a scale the shape down such that it is the same size as the template. Then various orientations are compared and the smallest result is used.

Moments of inertia

While template matching seems to be a good way of determining what shape an object represented, the problem arose whenever images were not placed in a fixed orientation. For this reason, it was more appropriate to try identify signatures that a shape may have that is independent of its orientation. The chosen signature was "moments of inertia".

While "moments of inertia" were the prefered method used in the previous years shape challenge, it was seen as a good method since it has been proven to work. This has been the prefered method as it was the UNSW Robocup 2001 team. The method used is shown in Section 9.3.4

Other methods

Other methods were examined and is also detailed in later sections.

9.2.5 Discussion

Given the task, various methods were explored. While this does not fully complete the discussion of methods examined, they provide a subset of what was examined, and what could be applied in future revisions of competition guidelines and challeges.

The methods explored here, while in the end was not the selected method. The combination of using Fourier transform to the "enhance" the image (producing a rotation invariant and size invariant version of the original image), then a simple pattern masking and then calculating the moments of inertia were quite effective methods. Despite the method being effective, and proven to work, it did however the implementation exhibited occasional cases where the Fourier transform stage failed for reasons that were not determined.

The anomaly produced incorrect scalings of images, where approximately 1 frame out of 20 frames from a streamed set of images from a stationary camera were used. This variation in performance made it a unsuitable condidate for competition.

Besides the anomly, advantages of the method include:

- Fourier transform could detect checker pattern even when robot was at it's start location (i.e. middle of the playing field) and looking at the shape (i.e. end of the field).
- Shapes always placed in a set orientation and scale. Easy to identify.
- Fast to process. Fourier transform is produced using FFT (see [JC65]). Pattern masking and shape signature calculation can be performed using a linear time complexity algorithm.

Problems include:

- Cannot detect more than one shape at a time, if two shapes are present in an image, unpredicatable behviour results.

- Results heavily affected when large areas of high contrast are in images. Affects the frequency information from Fourier transform.
- Weird anomly, until resolved, makes in unsuitable for competition.

Thanks must be provided to Alan Blair (blair@cse.unsw.edu.au) for providing ideas for Fourier transform.

9.3 Challenge 1 Approach Two(Competition)

Section author: Benjamin Leung

9.3.1 Literature Survey

There are already a number of algorithms developed for extracting checkerboard patterns out of an image using corner detection/ edge detection methods. An example of current research into this subject matter can be found at `http://www.stanford.edu/~mykel/chess/corner.htm`. These algorithms works on raw images and are quite computationally expensive, working on the assumption that there are no noise in the background. These technique detects checker boxes by detecting the corners of the boxes. Since rUNSWift was already investigating a image based approach - using fourier transform, checker box extraction using blobs was investigated as an alternative approach.

Moment invariant, which used as the signature for classification in this challenge, is a standard image processing technique. There are large volumes of information about moments, the particular book used by rUNSWift for reference is [RCG]. This books covers the equation and the calculation of the moment signatures. Since moment signatures were used last year in the challenges in with a similar goal, their code was also consulted.

9.3.2 Overview

The following section outlines the solution eventually used in the Fukuoka Competition.

In order to recognize the shape, the following steps were carried out:

1. Checkboard/Shape extraction from frame;
2. the shape is then painted on a canvas;
3. Signature generation of the shape on the canvas; then
4. Pass signature into a decision tree trained using samples.

9.3.3 Feature Extraction

The first step of the challenge is to extract the checker box pattern out of the image. The goal of this process is to extract a shape that resembles as close as possible the shape physically placed on top of the landmark.

The method of feature extraction used works at the blob level, i.e. as oppose to the image level where RGB or other direct values. In order to achieve that, a special calibration was made to allow detection of black colour. For the purpose of the challenge, black colour is represented on the colour plane as red. After blob formation is complete, red blobs will be where there are black. Working with this array of red patches, the extraction of the checkerboard can begin.

Firstly, all blobs too big or too small are discarded - the size of the checker boxes can be approximated because the distance to the beacon is known and the size of the checker box is invariant. Basically, the algorithm begins with the red blob closest to the top of the appropriate landmark and then using that as a seed, search for neighboring checkers and then neighbors of those neighboring checker boxes. After the blobs have been filtered, the blobs are classified as to whether they belong to the checkerboard pattern or not. The result of applying the algorithm can be seen in Figure 9.22.

```
if(visible vobYellowGoal or vobYellowRightBeacon or
        vobYellowLeftBeacon) {

    int startIndex = -1;
    int left = 0, right = 0, top = 0, bottom = 0;

    if (landmark.cf > 0)
    {

        Queue* pq = new Queue();

        // find the closest red blob to the bottom of the current landmark
        for(int i = 0 ; i < blobs.count(color); i++)
        {
            BlobInfo &info = blobs.getBlob(color,i);

            if (distance of blob to the top of the landmark
                    <= shortest detected distance &&
                 blob width >= 3 && blob height >= 3)
            {
                make this blob the seed;
            }

        }

        if (startIndex != -1)
        {
            push this blob onto the queue;
        }


        while(the queue is not empty)
        {
            seed blob = head of the queue;

            pop the queue;
```

```
avg_area += cur.area;

int maxArea = 2.0 * cur.area;
int minArea = 0.5 * cur.area;


for(each red blob on the screen)
{
    if the current blob has already been included
    {
        continue;
    }
    // if the blob is too far away
    if (distance between the centers of seed blob and current blob >
        (4 * 2 * area of the seed blob))
    {
        continue;
    }
    // if the blob is too big
    if (current blob width >
          seed blob width + MIN(2.0*(current blob width),10))
    {
        continue;
    }

    if (current blob height >
          seed blob height + MIN(2.0*(current blob height),10))
    {
        continue;
    }

    // if the blob is too small
    if (current blob width < MAX(0.5 * (seed blob width), 2))
    {
        continue;
    }

    // if the blob is too small
    if (current blob width < MAX(0.5 * (seed blob height), 2))
    {
        continue;
    }


    // if the aspect ratios are wrong (not square enough)
    if (current blob height / current blob width < 0.75 ||
        current blob height /  current blob width > 1.5 )
    {
        continue;
    }
```

```
                //if the blob satisfy all the conditions, that means it is a
                //neighbouring checker box to the current seed checker box
                // enqueue this blob to search for its neighbors later

                enqueue this blob;
            }
        }
    }
}
```

Given the collection of blobs that constitute the checkerboard pattern, the shape can be reconstructed. A number of methods were attempted, but most of them failed to provide a sufficiently distinctive shape for signature generation to work properly.

The basic idea of all the method is to reconstruct the shape using information retrieved from the blobs (checker boxes of the shape). Information such as blob size, blob centroid coordinate are all used in an attempt to rebuild the shape on a canvas.

Approach One

The first method attempted was to distinguish the shapes based on the area and position of the blobs (which represent each checker box). So the canvas consist basically of all the blobs that are believed to make up the checker box. The signature generated from this collection of blobs failed to provide a consistent and unique identify for the shape. This is mainly due to the poor quality of the camera and the sensitive colour classification scheme used. Often the size of the blobs and position of the blobs between different frames within the a space of 1 sec vary sufficiently to cause the classification tree generated to fail. For a visual representation of the canvas, see Figure 9.23.

Approach Two

In an attempt to eliminate problems caused by varying blob size that cause signatures to fluctuate and errors due to perspective, each blob is given the value of the average of the total areas of the blob or given a constant size of 1 (see Figure 9.24). This attempted to remove the weighing of different sized patches that influence signature generation. Different size blobs occurs when the shape is views from an angle; i.e. when the Aibo is looking up at the shape. The checker box closer to the camera appears much taller and larger than the boxes near the top of the shape due to the vanishing point problem. As such, the mass of the shape will always seem to be on the bottom - the shape seems to sag.

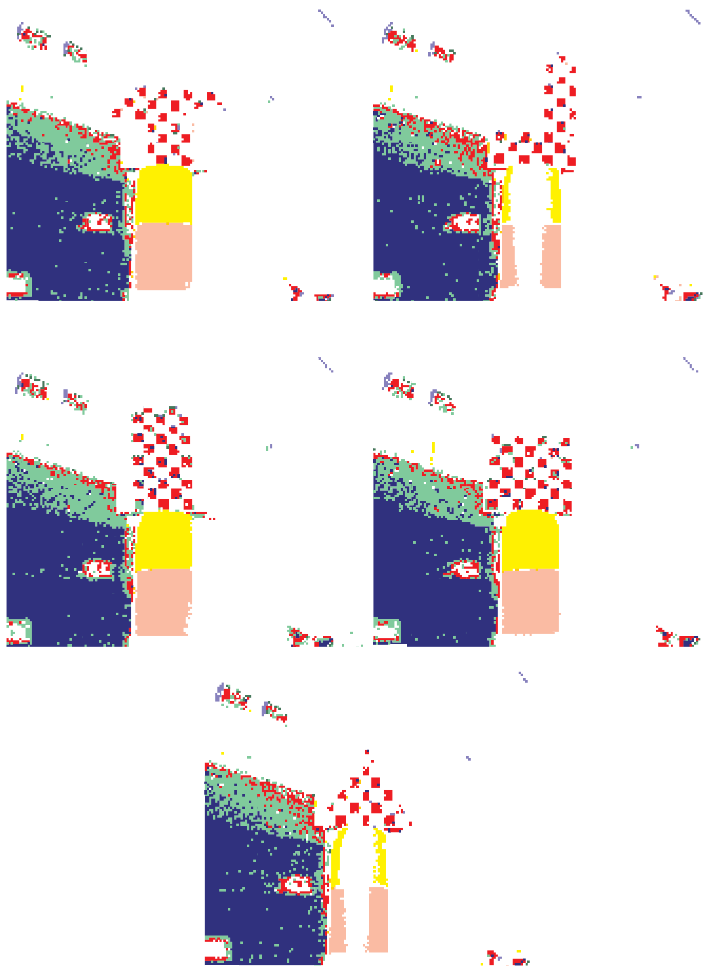


Figure 9.22: The C Plane view of the shapes from the Aibo camera. The bound box includes all the blobs which the above algorithm classifies as being part of the checker box pattern. From the top to bottom, left to right: T Shape, L Shape, Square, Rectangle and Triangle

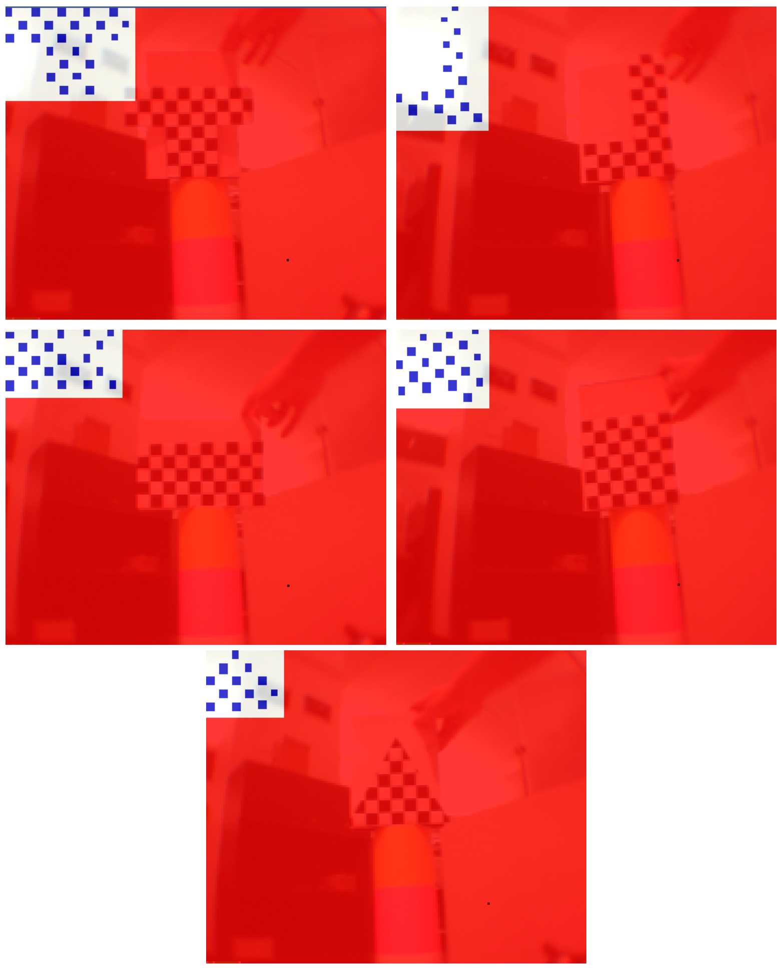


Figure 9.23: Using the original blob location and size to calculate the signature. The moment invariant is calculated for the white area on the top left corner of the picture. From the top to bottom, left to right: T Shape, L Shape, Square, Rectangle and Triangle

Using single pixels to represent a blob gives a much more consistent signature for each shape, but still failed to provide sufficient distinction between similar shapes. In particular, the T Shape and the Triangle is still troublesome - they only differ by 2 checker boxes which often confused the learning algorithms.

Final Approach

Since reconstructing just the black areas of the checker box patterns has failed to provide distinct shape signatures, feature extraction must extract the shape that the *Cardboard* cut out represents (i.e. the outline of the shape, the combination of both black *AND* white boxes). Given that its not feasible to detect white colour; since a lot of the field environment consist of white, e.g. the fence, the edge of the field etc., a more advance method must be used.

The method used to generate the outline of the shape is to draw a line between the centers of adjacent blobs. What this produces is then a mesh whose outline is a rough representation of the cardboard cutout (see Figure 9.25). This method yield complete distinctions between all the shapes, solving all the problems of the above approaches. To take this one step further, the space bounded by this outline is filled using a spacing filling algorithm (seed filling) - producing a shape that is almost identical to the cardboard cut out.

9.3.4 Signature Generation

Once the checker box pattern is extracted from the background, a signature must be generated in order to classify the detected shape. Moment invariants based on the extracted shapes were used to feed into the c4.5. Other features such as convex hull, perimeters, other ratios were also tried but did not improve the performance of c4.5.

Moments Invariants

Moment invariants are properties of connected regions in binary images that are invariant to translation, rotation and scale. They are standard techniques used in shape classification and part recognition. To convert our extracted shapes into binary images, all white is set to 0 and all other colour is set to 1.

The seven translation, rotation and scale invariant signatures are based on central moments (which are only translation and scale invariant).

The central $\alpha\beta^{th}$ (i.e. translation invariant) moment is defined by:

$$M_{\alpha\beta} = \sum_{i=1}^{N} B_i (X_i - \widehat{X})^{\alpha} (Y_i - \widehat{Y})^{\beta}$$

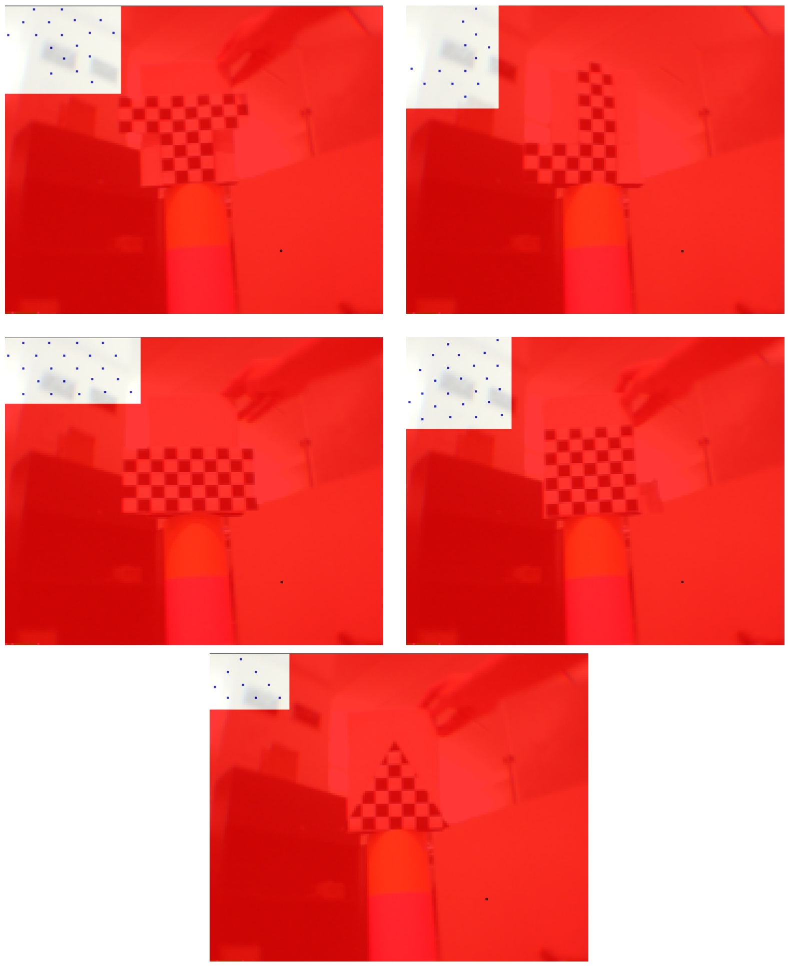


Figure 9.24: Assigning each blob size 1 and using their centroid to calculate the signature. The moment invariant is calculated for the white area on the top left corner of the picture. From the top to bottom, left to right: T Shape, L Shape, Square, Rectangle and Triangle

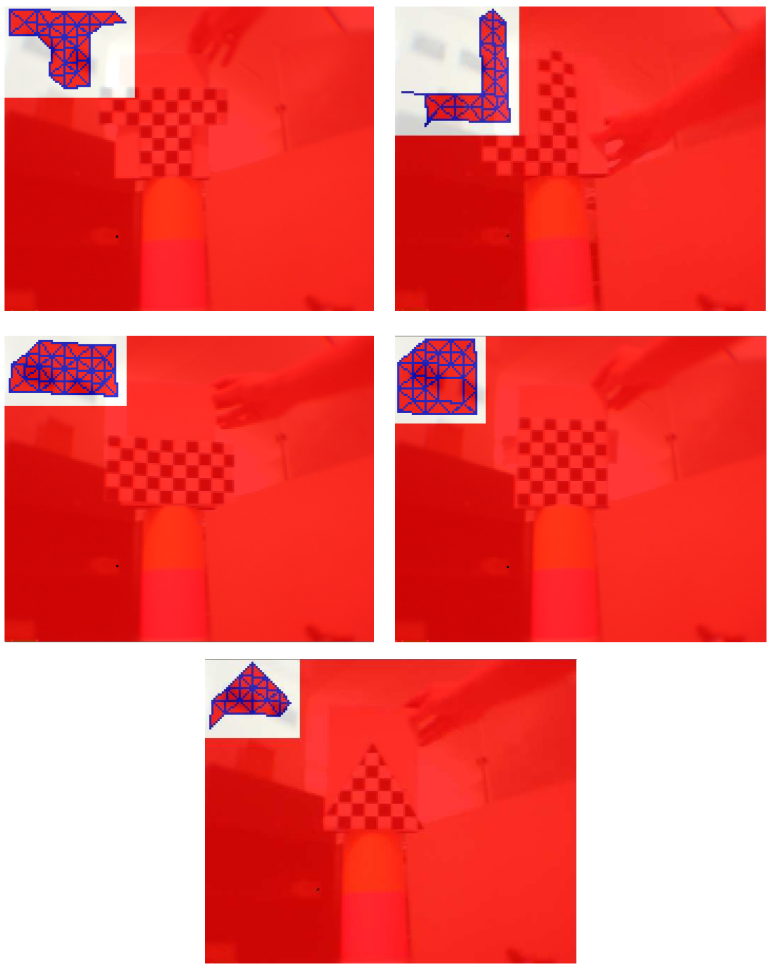


Figure 9.25: Forming a mesh between all the centroids and filling in the space between gives the best reconstruction of the shapes. The moment invariant is calculated for the white area on the top left corner of the picture. From the top to bottom, left to right: T Shape, L Shape, Square, Rectangle and Triangle

where $\widehat{X}$ is the average X position:

$$\widehat{X} = \frac{1}{A}\sum_{i=1}^{N} B_i X_i$$

and $\widehat{Y}$ is the average Y position:

$$\widehat{Y} = \frac{1}{A}\sum_{i=1}^{N} B_i Y_i$$

and B_i is the binary (0 or 1) value of the pixel at position i. The area A is the number of 1 pixels in the binary image. It is by subtracting the center-of-mass during the computation that these values are made translation invariant - as a part translates, so does the center of mass. But, the relative position of the part and its center-of-mass remains constant.

From these moments, we can calculate a scale invariant moment:

$$U_{\alpha\beta} = \frac{M_{\alpha\beta}}{A^{(\frac{\alpha+\beta}{2})+1}}$$

If a part doubles in size, then the value of M increases in proportion to its area and to the powers of the two moment terms. So, the denominator of the scale-invariant function increases by the same amount as M, and hence the ratio is scale invariant.

Via the theory of algebraic invariants (G.B.Gurevich. Foundations of the Theory of Algebraic Invariants. The Netherlands: P. Noordhoff, 1964), it is possible to find certain polynomials of M_{pq} that remain unchanged under rotation. From the $U_{\alpha\beta}$ we can calculate some rotation invariant moments:

$$I_1 = U_{20} + U_{02} \tag{9.3}$$

$$I_2 = (U_{20} - U_{02})^2 + 4U_{11}^2 \tag{9.4}$$

$$I_3 = (U_{30} - 3U_{12})^2 + (3U_{21} - U_{03})^2 \tag{9.5}$$

$$I_4 = (U_{30} + U_{12})^2 + (U_{21} + U_{03})^2 \tag{9.6}$$

$$\begin{aligned} I_5 = {} & (U_{30} - 3U_{12})(U_{30} + U_{12})[(U_{30} + U_{12})^2 - 3(U_{21} + U_{03})^2] + \\ & (3U_{21} - U_{03})(U_{21} + U_{03}))[3(U_{30} + U_{12})^2 - (U_{21} + U_{03})^2] \end{aligned} \tag{9.7}$$

$$I_6 = (U_{20} - U_{02})[(U_{30} + U_{12})^2 - (U_{21} + U_{03})^2] + 4U_{11}(U_{30} + U_{12})(U_{21} + U_{03}) \quad (9.8)$$

$$I_7 = (3U_{21} - U_{03})(U_{30} + U_{12})[(U_{30} + U_{12})^2 - 3(U_{21} + U_{03})^2] - (U_{30} - 3U_{12})(U_{21} + U_{03})[3(U_{30} + U_{12})^2 - (U_{21} + U_{03})^2] \quad (9.9)$$

The combination of the above seven values are unique given a particular shape - independent of the rotation, scale and translation of the shape. The above invariants are calculated on the white area of the canvas.

9.3.5 Shape Detection

Using c4.5 learning algorithm on large samples of signatures, a classification tree was generated to determine the shape when given a set of invariants. A number of different trees were generated and tried. Initial attempts used large number of samples (3000 different samples) to generate the decision tree but the resultant classifier seem over-filled the data and also the respective error. The final decision tree used in competition shown below was modified from one generated using only 500 samples:

```
if (I4 <= 0.001192) {
    if (I1 <= 0.199769)
    {
        if (I3 > 0.000963)
        {
            return Triangle;
        }
        else
        {
            return Square;
        }
    }
    else
    {
        if (I3 <= 0.002887)
        {
            return Rectangle;
        }
        else
        {
            return TShape;
        }
    }
```

```
}
else
{
    return LShape;
}
```

It is interesting to note that only invariant 1, 3 and 4 are used to classify the shapes.

Given that there are random noise and individual decisions can be incorrect, the shape challenge uses a simple voting system to confirm the output of the shape detection algorithm. Each frame represents 1 vote for a particular shape and when 75 votes has been taken, the final decision is the shape with the most votes. More complicated voting systems were tried - using confidences, multiple votes etc, but they proved no more effective than this simple method. 75 votes were chosen because of the 3 minute time constraint of the challenge. 75 frames (roughly 15 seconds - depending on noise, see 9.3.7) is the maximal time the Aibo can spend analyzing a shape in order to complete the challenge within 2 minutes and 30 seconds. A majority of the time is used for localization and position the robot so that it gets a square and front on view of the shape.

9.3.6 Locomotion and Localization Issues

Given the sensitivity of the shape detection algorithm, it is extremely important that the Aibo get a good view of the shape - a nice front on view and of the whole shape. Given the elevation of the shape, the current pwalk stance often does not give a flat enough view of the shape. So when the performing shape analysis, the Aibo changes to a higher stance by straightening its front legs.

The locomotion sequence of challenge 1 is shown in Figure 9.26. The Aibo starts at the center of the field in the beginning of the challenge, and visits each of the shape starting from its left. The optimal image analysis distance is just outside the wag-the-tail area, about 80cm from the shape. At this distance, the robot changes stance and performs image analysis. After this, it resumes walking stance and move into the wag-the-tail area.

After the first shape, it returns to position two to re-localize. It then visits the shape above the goal. The elevation of the shape on top of the goal is not as great as the beacons, so a higher stance is unnecessary. Also, given that the shape is at the back of the goal, the Aibo will already be in the wag the tail area during the image analysis stage - so moving in is not required.

After shape two, it once again moves back to position two to re-localize. It then visits the final shape in the same fashion as shape one.

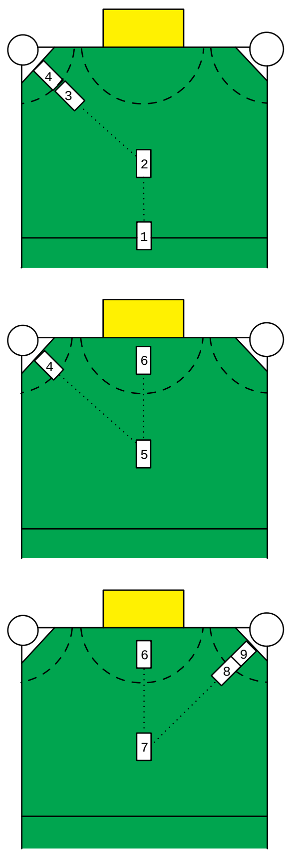


Figure 9.26: The way in which the Aibo visits the shape in challenge 1

9.3.7 Limitations

The method used to extract the checker box blobs from the image is quite slow. The algorithm contains nested for loops (complexity n^2) - which becomes exponentially expensive as the number of blobs increases. In extremely noisy environment, the extra processing can slow down the frame rate to around 10 frames per second. In the Robocup lab environment, the frame rate is usually around 20 frames per second. While this is not important for the challenge, if the goal ultimate goal is to replace the beacons with checker boxes, a much faster and less computationally expensive method must be found to detect checker boxes to keep the robots being responsive during the game.

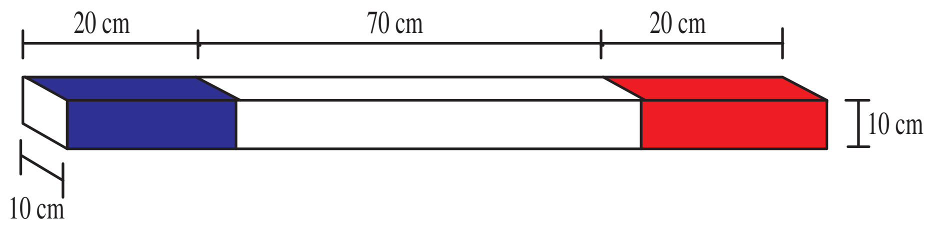
9.3.8 Competition Results

rUNSWift recognized only two out of three shapes, coming second in this challenge. CMU, whom came first, also only recognized two shapes but completed it in a shorter time.

Team	Cleared Steps	time 1	time 2	time 3	Rank	Points
CMU-Pack'02	**2**	**24.60**	**49.78**	**-**	**1**	**19**
rUNSWift	**2**	**40.48**	**84.60**	**-**	**2**	**18**
UW Huskies	**2**	**33.87**	**103.84**	**-**	**3**	**17**
NUbots	1	22.08	-	-	4	16
RoboMutts++	1	32.34	-	-	5	15
Baby Tigers	1	44.74	-	-	6	14
German Team	1	55.88	-	-	7	13
Upennalizers	1	89.72	-	-	8	12
Wright Eagle	1	97.86	-	-	9	11
ARAIBO	0	-	-	-	10	7
Les 3 Mousquetaires	0	-	-	-	10	7
Team Sweden	0	-	-	-	10	7
McGill Red Dog	0	-	-	-	10	7
Essex Rovers	0	-	-	-	10	7
ASURA	0	-	-	-	10	7
Cerberus	0	-	-	-	10	7
SPQR	Disqualified	-	-	-	-	-
Borregos Salvajes	Disqualified	-	-	-	-	-
Yellow Jackets	Disqualified	-	-	-	-	-

9.3.9 Conclusion

The code used in competition was quite thoroughly tested in the labs prior to departure for Japan. Combinations of different sized shapes and orientations were tried. The success rate of accurately recognizing Square, Rectangle and L Shape is about 95 percent. The most troublesome shapes is T Shape and Triangle, even then accuracy is about 80 percent. In the competition, it is the Triangle that rUNSWift failed to recognize in the competition. The triangle was placed with its top pointing to the right, and it was misclassified as a T Shape. The most likely cause of this misclassification is probably due to the Japanese kanji characters on the poster which Sony used to put behind the shapes. The characters were black and their square shape made them very similar to the checker boxes. Since the T Shape and the Triangle only differ by a few black squares, these characters made the shape seemed like T. Interesting note is that no other team got this shape correct either.



9.4 Challenge 2 Collaboration Challenge

Section author: Benjamin Leung

9.4.1 Overview

In this challenge two robots - one blue and one red are required to *collaborate* together in order to move a foam bar around the field and to the appropriate position.

The Bar

The bar is a $10cm \times 10cm \times 90cm$ rectangular prism made out of foam. Its colour is white and each end is covered by a sticker in the colour of the robot uniforms. When being pushed by the robots, it generally slides more often than rolls.

9.4.2 Rules

Starting Position

Please refer to Figure 9.27 for clarifications.

- Each dog is placed behind the half way line facing coloured ends of the bar (red facing red and blue facing blue.
- The bar is placed half way between the half way line and the end of the field.
- Points will be awarded each robot that touch the bar
- Third point will be awarded when one end of the bar passes the center line
- Fourth point will be awarded when the other end of the bar passes the centerline

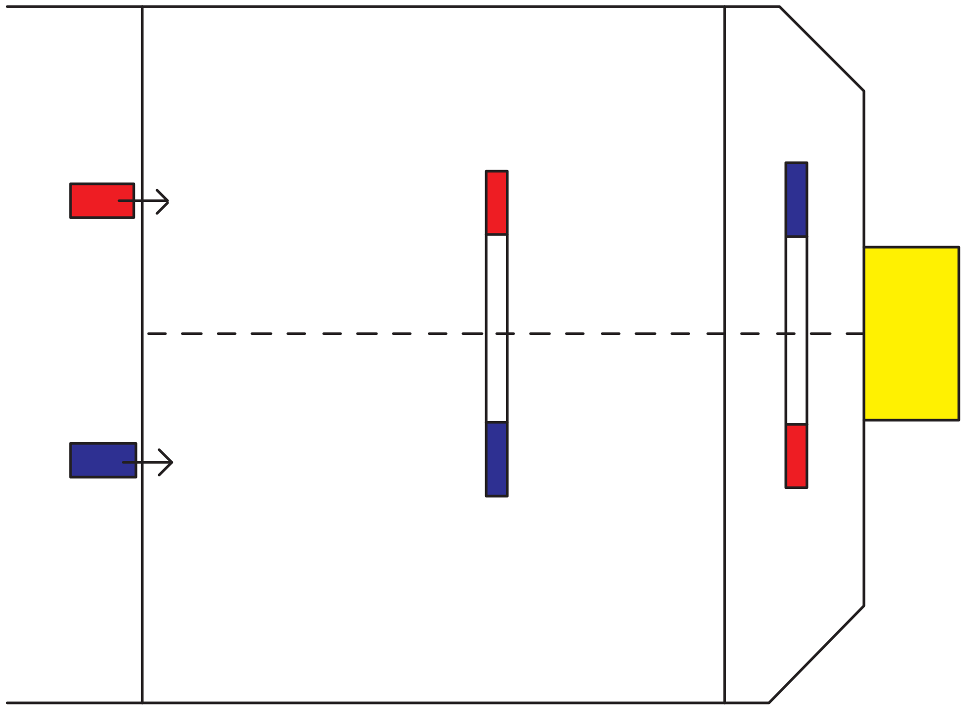


Figure 9.27: The starting position of challenge 2. There are two bars shown on the picture, the left is the position of the bar at the beginning of the challenge, and the right is the final position of the bar at the end of the challenge.

- Fifth point will be awarded when the bar is within the penalty area.
- Each dog starts in front of their own colour ends;
- The bar is place half way between the half-way line and the end of the field;

9.4.3 Sub Strategy

Instead of using wireless to enable communication between the two dogs, it was decided to use visual triggers to make state changes. The decision not to use wireless was due to the following factors:

- wireless was proven to be less than 100% reliable;
- wireless may fail to operate during the course of the challenge and;
- state changes using unreliable communication can lead to deadlock;

Using wireless would introduce a level of uncertainly that would require a lot of run-time checks.

Specialized Vision

In order to correct determine the distance and heading of the coloured end of the bar, changes needed to made to the vision module of the code. As such, the robot detection of VisionCortex was modified to detect the ends of the bar.

The tradition method of using bounding box size cannot be used in this case, due to the same problems encountered in robot detection. The non uniform shape of the rectangular prism when viewed from different angles made impossible to use the bounding box approach(see Figure 9.28).

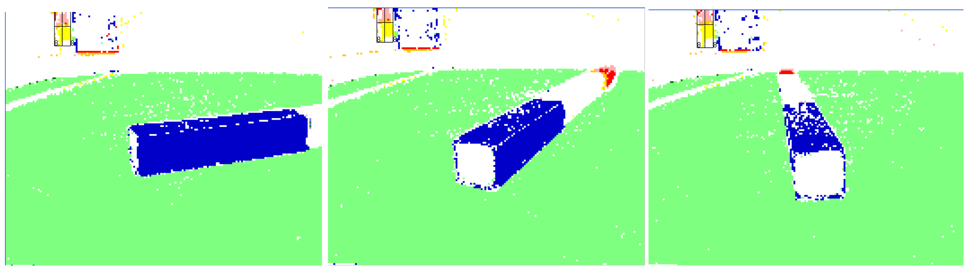


Figure 9.28: The changing area of the bounding box around the end of the bar when viewed from different angles makes it impossible to use this property for distance calculation.

Detecting the distance of the chest of the robot to the closest point of the coloured end of the bar uses the deviation method identical to the method used in robot detection (see Figure 9.29). The distance, given by equation 9.10 is much more reliable when compared to distance given for robot recognition because:

- the bar is static;
- the movement of the dog is much smoother in this challenge and thus there is less shaking of the camera and;
- the coloured end of the bar makes contact with the ground, giving a much more accurate triangle.

The distance of to the coloured end of the bar is determined by the following equation:

$$distance = \frac{15.0}{\tan(effective\ tilt - beta)} \tag{9.10}$$

Moving the Bar

There are several different ways of moving the bar around the field. The robot can either push the bar using its front paw as it walks forward OR it can push

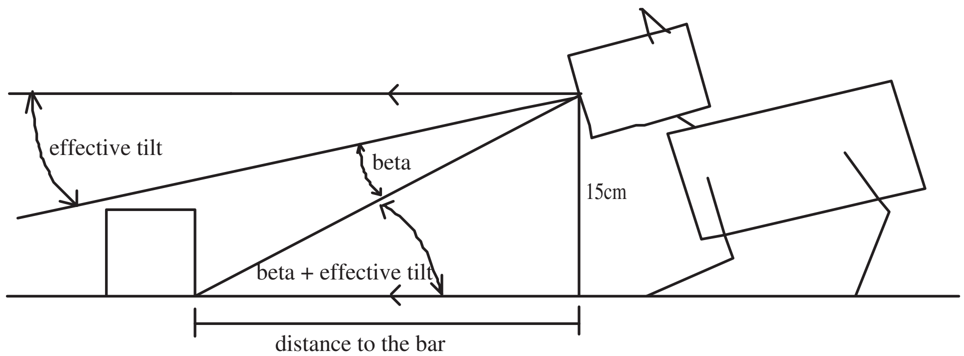


Figure 9.29: Using simple trigonometry with the outlined dimensions resulted in equation 9.10

the bar by walking side ways. In this particular challenge, the robots push the bar by walking side ways. This was because:

- the robot walks faster when moving sides ways when compared to walking forward. (see Figure 9.30);
- by making the robot face the end of the other bar (the center of circular path) distance calculations becomes much more accurate; and
- the small area of contact with the bar and large space between the paws when pushing makes it less reliable than using the entire side of the body. When orbiting the other end of the bar, often only 1 paw connects with the bar due to the circular nature of the path.

9.4.4 Strategy

Figure 9.31 to Figure 9.37 outline the final locomotion sequence for challenge one. The dotted lines represent the path of movement while the arrows from the boxes represent the head direction of the Aibo.

Many different methods were tried, this final sequence proved to be the most robust and the fastest. A few observations made from using different sequences helped formulate the final solution:

- accuracy of localization deteriorates very quickly because the Aibo continuously focuses on the end of the bar. So all triggers should be relative to the robot itself (i.e. start next move when red end's heading is +60 degrees etc).
- try to keep the red robot and the red end of the bar together and the blue robot and the blue end of the bar together. This is so that the Aibo's do not get confused as to which one is the bar and which one is not.

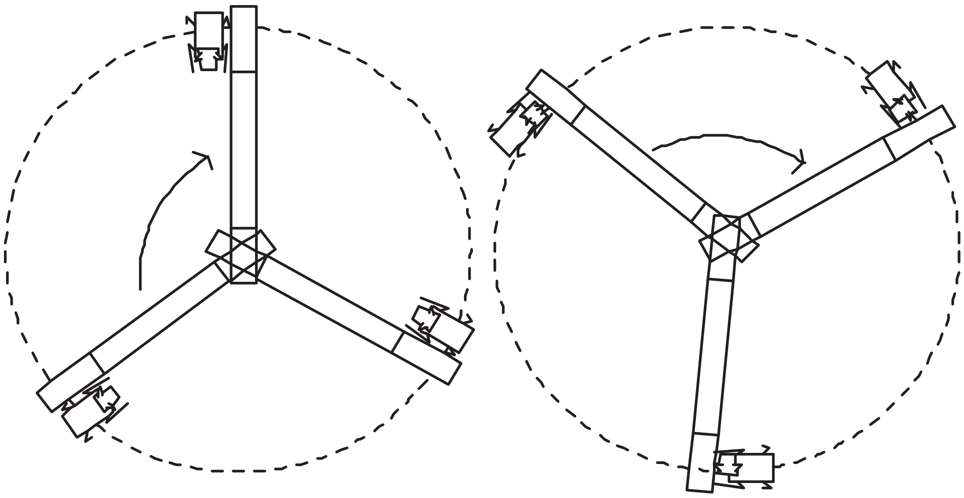


Figure 9.30: It only takes 9 seconds for a robot walking sideways to complete one revolution while it takes 15+ seconds for a robot walking forwards to complete the same task.

- Aibo walks faster sideways than forwards.

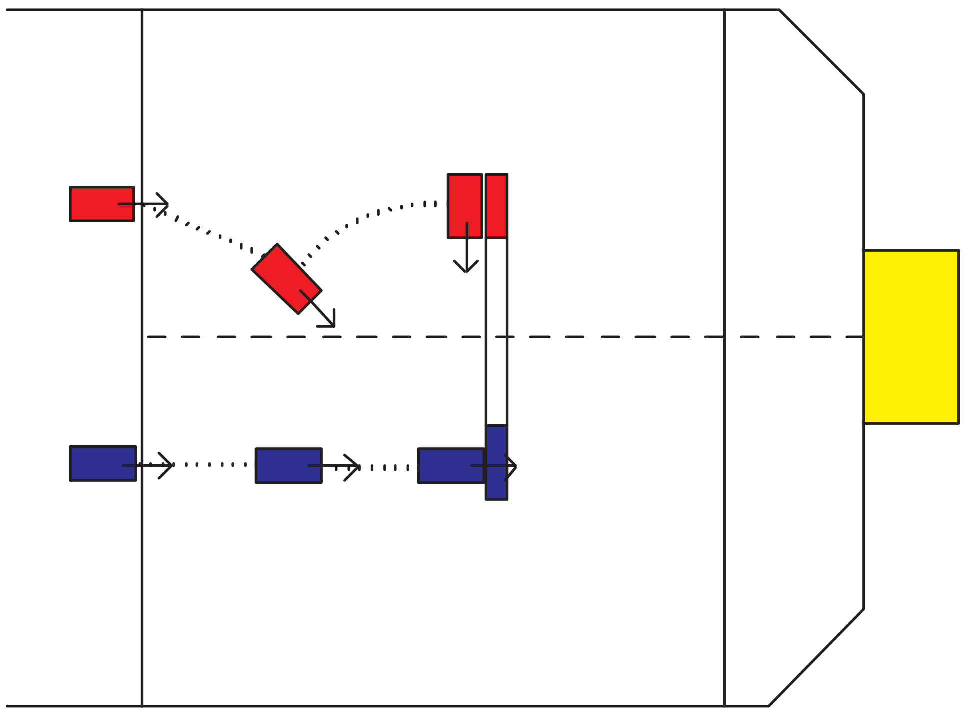


Figure 9.31: Step One.

The Blue robot looks at the blue end and walks until it touches it with its two paw. The trigger to stop is the tilt of the head when looking directly at the bar(stop when tilt ¡ -80). The Red robot also looks at the blue end of the bar and circles clockwise about it within a 70cm radius. It then touches the red end with the side of its body.

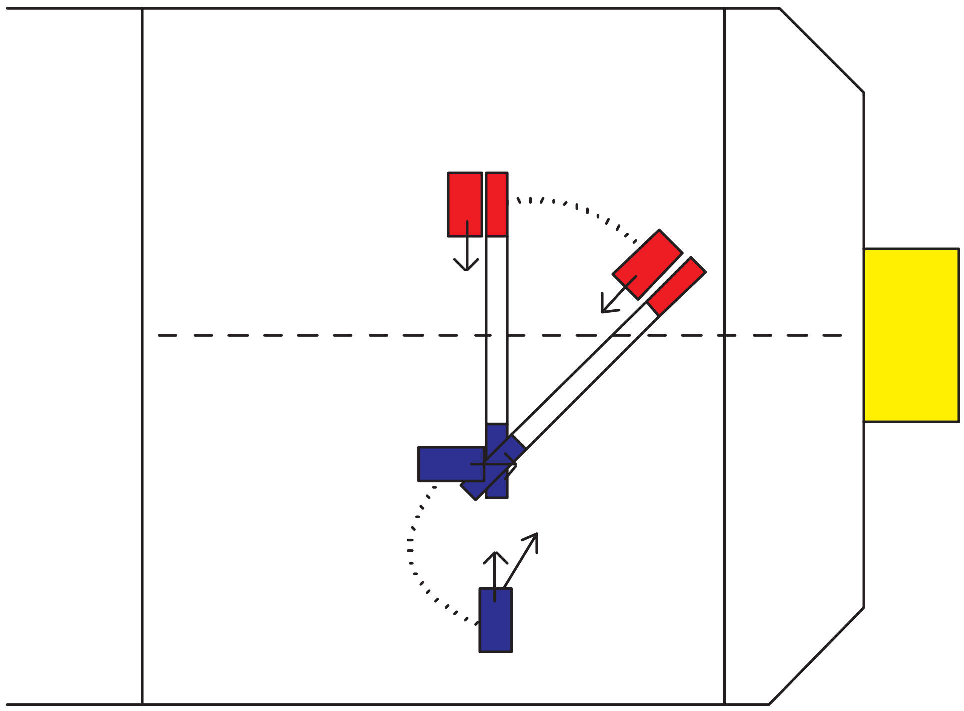


Figure 9.32: Step Two.

Red robot continues circling and pushing the bar. Blue robot backs off the blue end and circles about it anticlockwise at a distance of 50cm until it is parallel to the two ends of the field. It then changes its focus to the red bar and Red robot, awaiting the visual trigger.

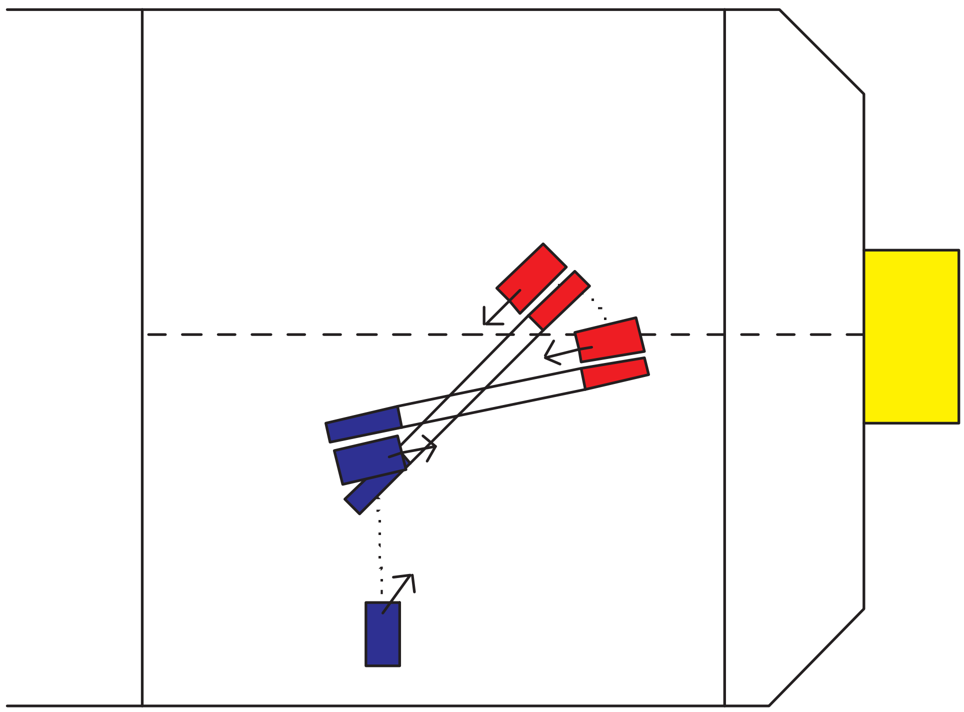


Figure 9.33: Step Three.

Red robot continues pushing the bar until a certain heading is reached (stop when robot heading ¡ 125). Red robot then stops and looks at the blue end of the bar. Blue robot waits until the heading of red robot relative to itself to be greater than a certain angle (star when red robot heading ¡ -20). It then begins to circle clockwise about the red end within a 70cm radius.

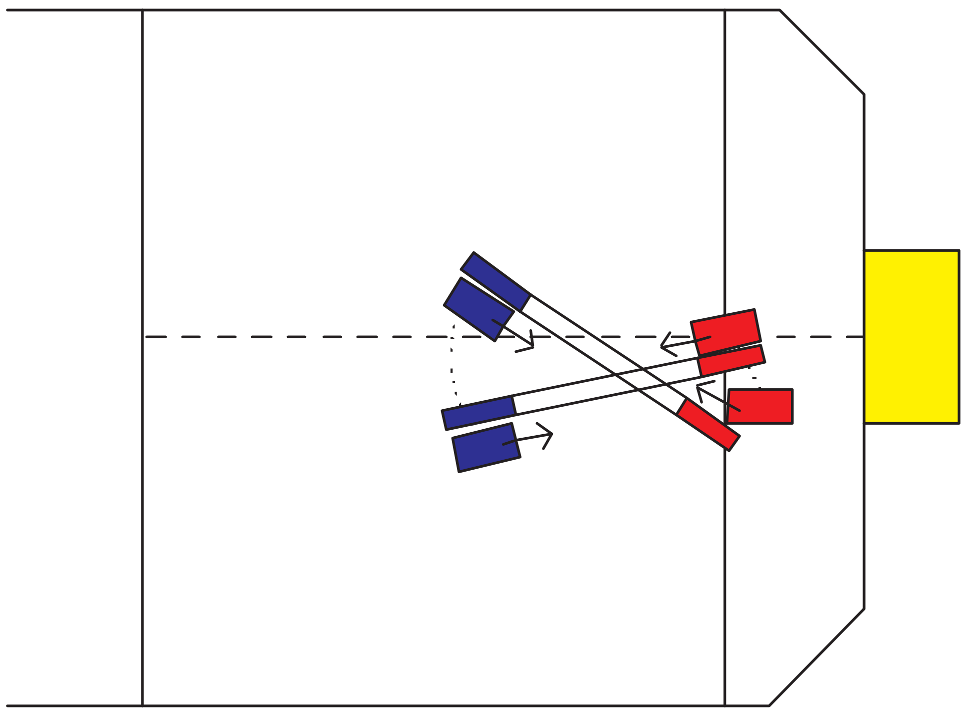


Figure 9.34: Step Four.

Blue robot continues to push the bar. When Red robot detects the heading of blue robot to be less than zero (start when blue robot heading ¡ 0), it strafes left, pushing the red end to help make the bar parallel, and also to get out of the way. Originally, the red robot backs off into a clockwise orbit about the red end - but the robot often gets suck on the goal corners.

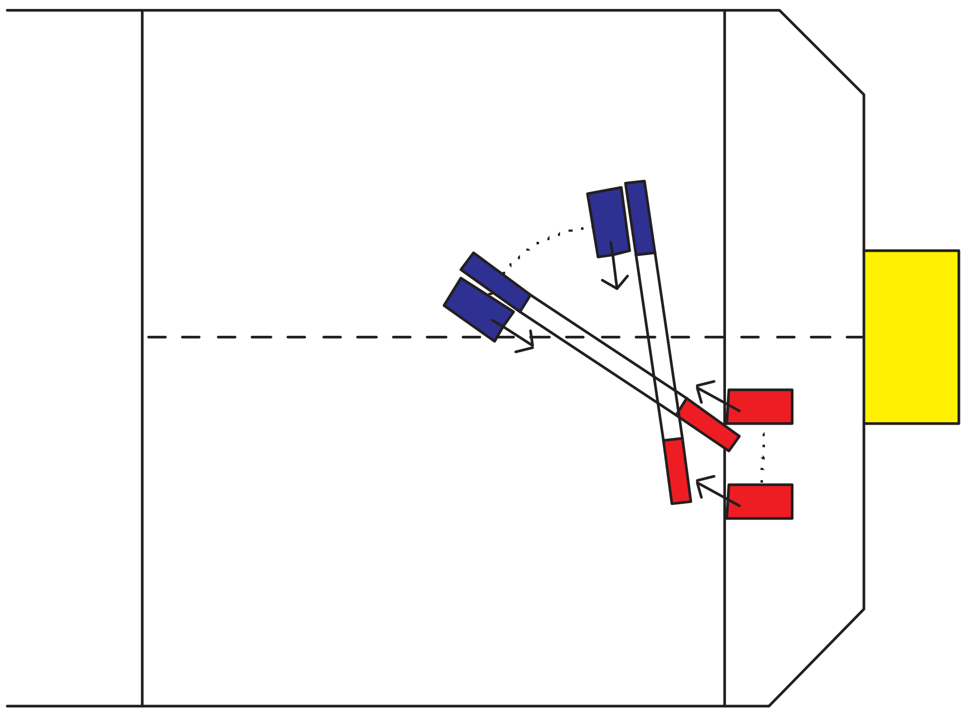


Figure 9.35: Step Five.

Blue robot continues to push the bar around. After strafing, red robot changes focus to the red end of the bar.

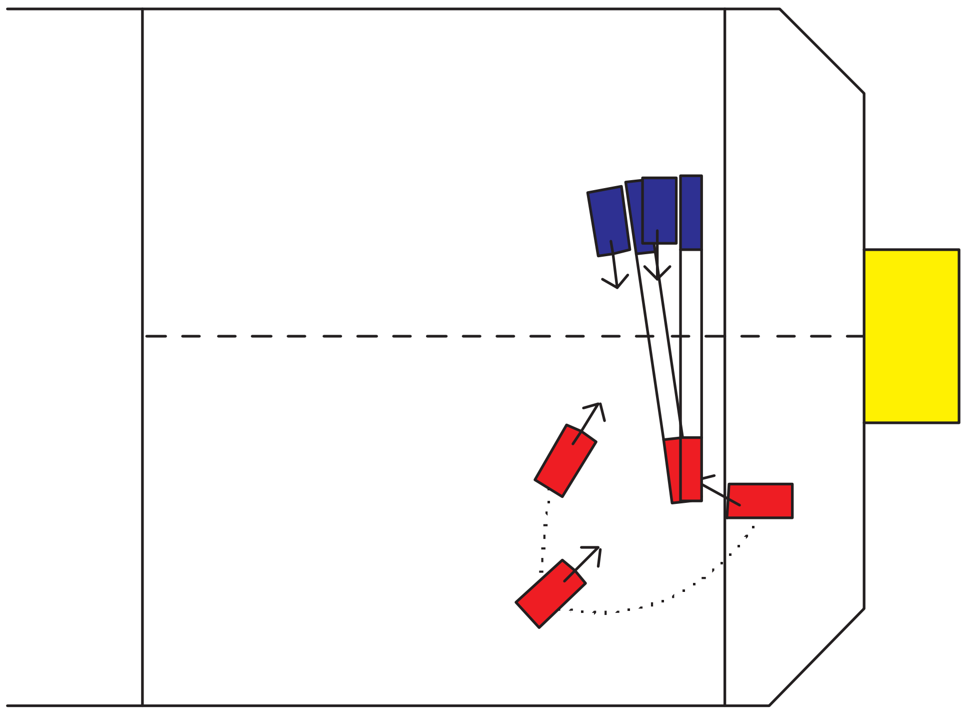


Figure 9.36: Step Six.

Blue robot continues to push the bar until its heading it parallel to the ends of the field (stop when robot heading = 180). It looks at the red end of the bar waiting for Red robot to touch it. It will always try to keep the heading of the red end of the bar relative itself as zero.

Red robot circles clockwise about the red end of the bar within 50 cm radius until a certain heading (stop when robot heading = -90). It then changes focus to the blue end of the bar, and then circles anti-clockwise about the blue end with a 70cm radius.

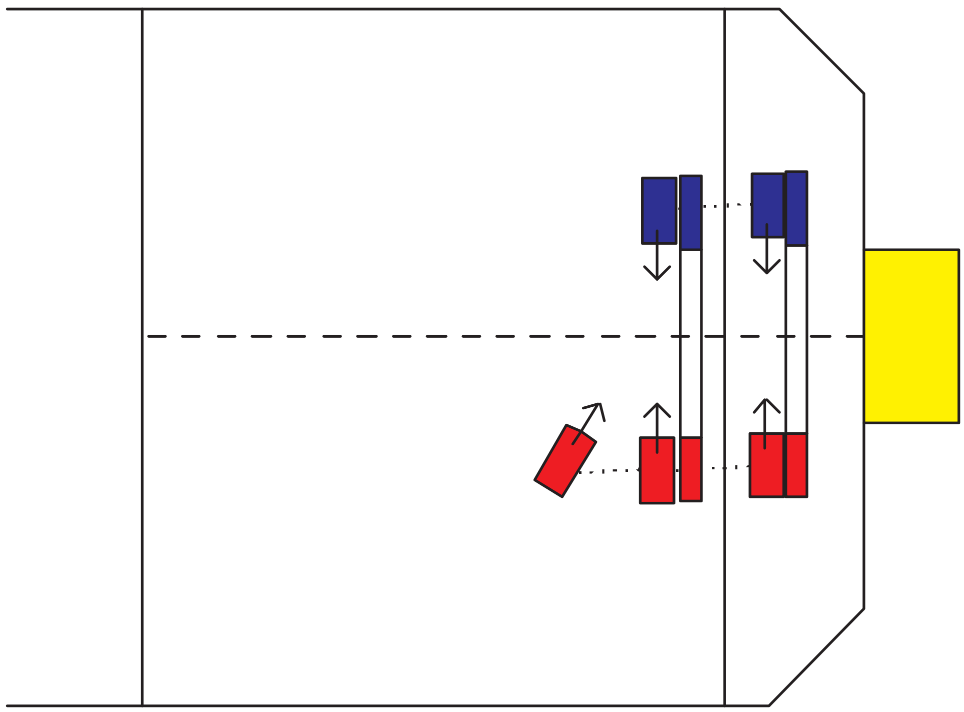


Figure 9.37: Step Seven.

The red robot pushes the bar, it triggers the blue robot to readjust its position. Red robot continues to circle the blue end at 70cm. As both robots try to circle about their opposite ends, because both ends are moving - the robots will adjust their paths to maintain their distance to the end of the bar. As a result, the overall effect is moving the bar horizontally pass the penalty line and into the goal area.

9.4.5 Competition Result

Team	Cleared Steps	time1	time2	time3	time4	time5	Rank	Points
rUNSWift	**5**	**7.69**	**9.76**	**16.96**	**30.23**	**44.09**	**1**	**19**
CM-Pack'02	**5**	**7.04**	**16.32**	**21.65**	**24.48**	**70.79**	**2**	**18**
Wright Eagle	**5**	**18.05**	**39.21**	**43.00**	**45.24**	**134.29**	**3**	**17**
NUbots	4	8.74	9.84	19.34	24.94	-	4	16
RoboMutts++	4	9.16	10.53	13.85	25.48	-	5	15
ARAIBO	4	7.15	12.46	31.23	50.21	-	6	14
Team Sweden	4	11.18	12.45	53.26	57.66	-	7	13
Upennalizers	3	10.15	14.77	22.27	-	-	8	12
SPQR	3	16.16	22.10	27.38	-	-	9	11
McGill Red Dogs	3	10.48	12.55	32.79	-	-	10	10
UW Huskies	3	28.04	42.23	78.40	-	-	11	9
German Team	3	30.38	36.34	51.54	-	-	12	8
Essex Rovers	3	14.04	26.63	101.10	-	-	13	7
Les 3 Mousquetaires	2	48.63	97.56	-	-	-	14	6
Baby Tigers	1	6.64	-	-	-	-	15	5
CeRed roboterus	1	12.43	-	-	-	-	16	4
ASURA	1	31.87	-	-	-	-	17	3
Borregos Salvajes	0	-	-	-	-	-	-	-
Yellow Jackets	0	-	-	-	-	-	-	-

9.4.6 Conclusion

The success of rUNSWift in this challenge is largely attributed to the deviation method of calculating distance to the end of the bar. Because distances were calculated accurately, this helped in preventing the bar from slipping when being moved by the robot and also allow the robot code to operate relative to the end of the bar. Some of the teams seemed to rely more on localization to move the bar around and while this also worked, it is clearly not as reliable or as fast as the method used by rUNSWift. The rUNSWift solution focused on moving the bar around as oppose to position robots accurately on the field.

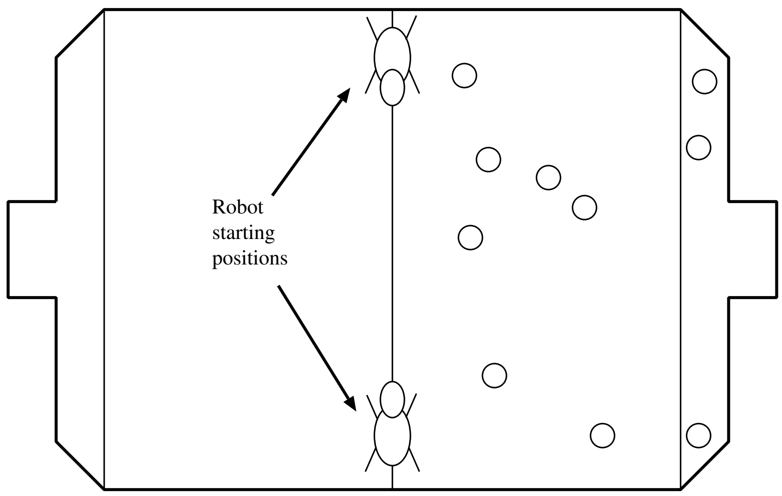


Figure 9.38: A sample initial configuration.

9.5 Challenge 3 Ball Collection Challenge

Section author: Min Sub Kim.

9.5.1 Challenge Overview

The ball collection challenge required two robots to "collect" ten balls by shooting them into either goal. The challenge had a time limit of three minutes, and teams were ranked according to how many of the ten balls they managed to put into the goals.

The initial configuration of the challenge saw the two robots at the center line, facing the middle of the field. The ten balls would be randomly placed on one half of the field; the exact placement was not revealed until the challenge was ready to begin, but the placement itself is the same for all teams. A sample initial configuration is shown in figure 9.38.

One issue that may arise is the possibility of the robot chasing a ball that is already in the goal. This issue was resolved as Sony decided that they would remove balls from the field as they entered the goal.

9.5.2 Strategy overview

The strategy that we employed for the ball collection challenge can be essentially divided into two parts: the strategy to be taken by a robot by itself, and the strategy that makes the robots act cooperatively.

Single robot strategy

This was the starting point of strategy development for the ball collection challenge. The single robot strategy is the essential core of the strategy, and looks at the challenge as a repetition of a sequence of steps. These steps are designed to reliably shoot one ball into the nearest goal, and thus can be simply repeated until either the time limit is reached or all the balls have been cleared.

The step sequence taken by a single robot is as follows:

- *Find a ball* - the robot spins towards some central position on the field, looking for a ball.
- *Approach the ball* - the robot approaches the ball it has found, tracking it as it moves closer.
- *Hold the ball* - the robot tightly holds onto the ball with its front legs, and then looks around to localise itself.
- *Aim for the goal* - the robot then turns to face the nearest goal, while keeping the ball held with its front legs.
- *Shoot* - finally, the robot shoots at the goal, using the powerful forward kick, and then the process is repeated.

The strategy began from this basic idea and was progressively tweaked to improve its efficiency and reliability. These modifications dealt with behaviour around the edges of the field, and using a variety of kicks depending on distance to the nearest goal.

The behaviour around the side edges of the field was the subject of much experimentation and optimisation. In general, balls placed on the edges of the field are usually the hardest to shoot into the goal, as they require a significant amount of work in order to dig them out. By contrast, balls placed out in the open are usually very easy to shoot into the goal, as the robot only needs to get to the ball, aim for the goal, and then shoot. Because the placement of the balls in the challenge were random, in order to ensure reliability, it was necessary to spend a lot of time making sure that the balls on the edges could be disposed with minimum delay. The edge behaviour modifications are as follows:

- *Approach* - If the ball found is on the edge of the field, then approach the ball perpendicular to the edge. This is because if the robot attempts to approach the ball at an angle that is close to parallel with the edge, it often gets stuck on the edge trying to center its body to the ball.
- *Aim for goal* - If the ball is on the edge, once the robot starts turning to aim for the goal, it often gets stuck trying to turn into the edge. This is because the turning action basically has the ball stay on the same spot on the field, while the robot turns around it. The first solution to this problem was to have the robot shoot along the edge instead of aiming for the goal. This way, the robot would shoot the ball into the corner, at which point it would roll out into the open near the front of the goal.

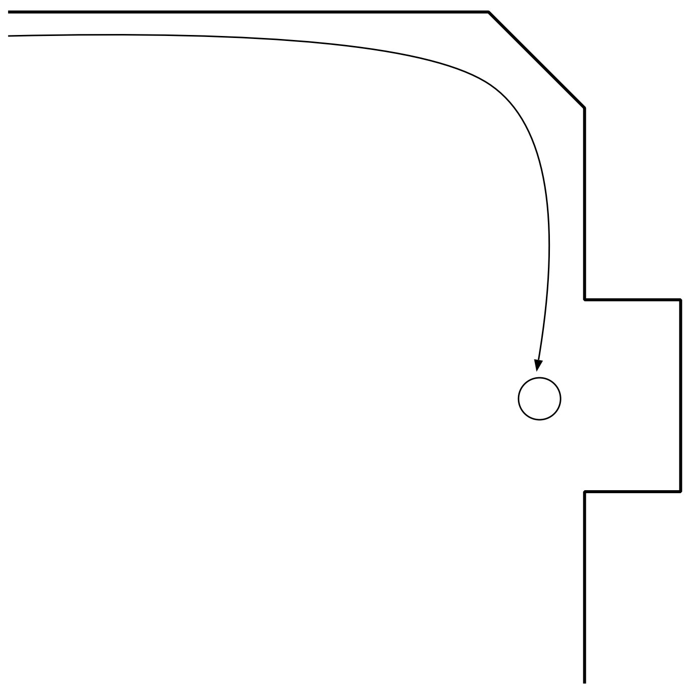


Figure 9.39: The intended effect of shooting the ball into the corner. The ball usually will not roll into the goal immediately, but usually stops a short distance in front of the goal, setting up an easy finish.

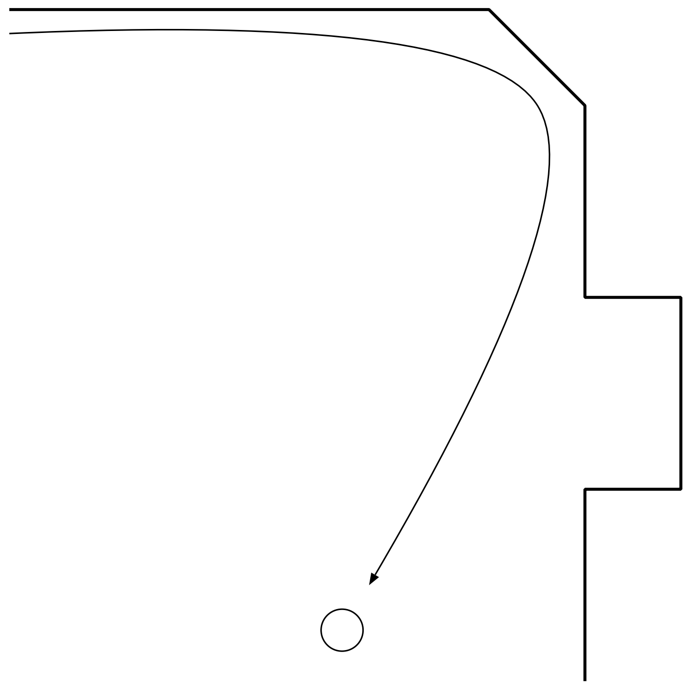


Figure 9.40: Shooting the ball into the corner has this drawback, where the ball sometimes rolls too far out, particularly if the chosen kick is too powerful. The robot then wastes a lot of time chasing the ball as it curves around the corner.

- *Walking sideways with the ball* - While the solution of shooting the ball along the edge seemed reasonable, it sometimes seemed to waste a lot of time as the robot would chase the ball after shooting it. The second solution devised was to walk sideways with the ball to move it away from the edge, so the robot would make room to turn to aim at the goal. To do this, the robot holds the ball, turns until it is facing parallel to the edge, and then goes into a steady sideways walk, with the head down to keep the ball in possession. Once it has moved far enough from the edge, it then holds the ball again, localises, and then aims and shoots at the goal.

In addition to optimising edge behaviour, a simple strategical improvement was to utilise a variety of kicks depending on distance to the nearest goal. This was mainly in order to deal with balls that were close to the corners. In this situation, the shooting angle to the goal is small, so if a powerful kick is used, it can easily shoot past the goal and end up in the opposite corner. The robot could then get stuck shooting the ball past the goal back and forth. To avoid this, the weaker chest push is used in such situations, so even if the ball doesn't go in the goal, usually it ends up right in front of the goal, which is then very easy to finish off.

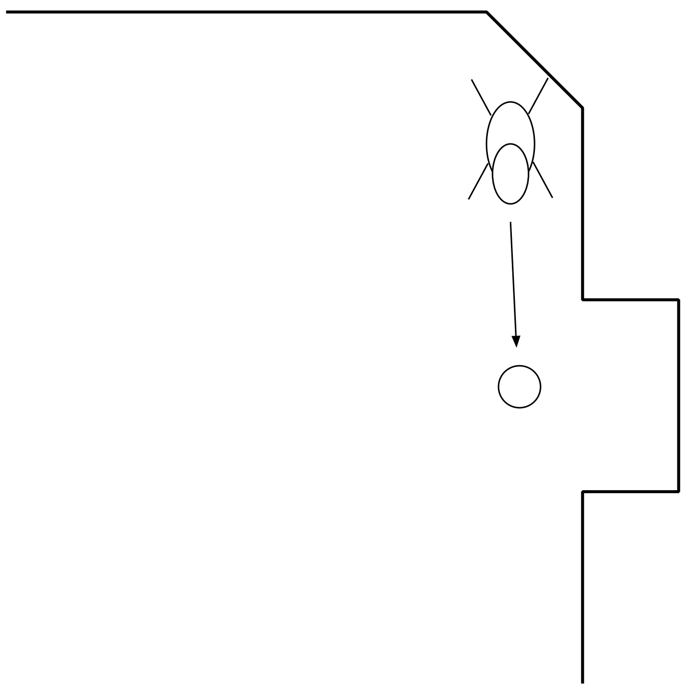


Figure 9.41: The intended effect of using the chest push at closer ranges. Because the shooting angle is too small, it is generally more efficient to use the chest push to place the ball near the goal mouth than to take a powerful shot.

Cooperative strategy

The development of the strategy for a single robot was relatively straightforward and required only moderate experimentation. Development of the cooperative strategy, on the other hand, was difficult and slow.

The cooperative strategy that was finally used in the challenge competition was actually quite simple, but sufficed nonetheless. There were many strategies that were discussed and tried, as well as many clever ideas that were never implemented but might have proved interesting to try out. The following is a discussion of these ideas, and an outline of the final strategy used.

Ally Avoidance

Ally avoidance was clearly the most fundamental cooperative strategy that needed to be used in this challenge. The two robots needed to ensure that they stayed out of each others way at all times, otherwise they would waste time and possibly even collect fewer balls than a single robot could by itself.

While the principle of ally avoidance seemed simple enough, the fine tuning of the skill required considerable experimenting to optimise. The first issue is the threshold at which the robots decide they are too close and could obstruct each other. If this range is too short, then it is more likely that the robots will collide, but on the other hand, if this range is too great, they may waste time unnecessarily trying to stay too far away from each other. The second issue that needed to be considered was the behaviour of the robots once the threshold was reached.

To deal with the first issue, the threshold was tuned empirically to a moderate level, leaning more towards being passive. The rationale for this decision was that the time lost by unnecessarily backing away would be relatively small compared to the time lost if the robots collided, or in the worst case, if their legs were locked in the collision. In addition, having different ranges for the two robots was also tried, with one robot being aggressive and the other being less aggressive, an idea borrowed from the two forward roles used in last year's strategy. This would help reduce the likelihood of a situation where both robots repeatedly head for the same ball and back off from each other, as the aggressive would get the ball and the less aggressive would retreat. This idea was eventually rejected as it had the drawback of the aggressive player moving in too close, particularly in situations where the less aggressive player was not in a position to give way.

The other issue to be dealt with is that of what the robots should actually do when the threshold is breached. The behaviour employed in match play under such situations can be summarised as backing away and moving into an attacking position, getting around the ball if it is visible. This strategy is not suitable for this challenge, since there are no opponents to worry about, and there are also multiple balls. Instead, a simpler approach was taken, with the robot walking backwards with a moderate degree of rotation, for a fixed duration. The robot would also keep its head high in the process, panning

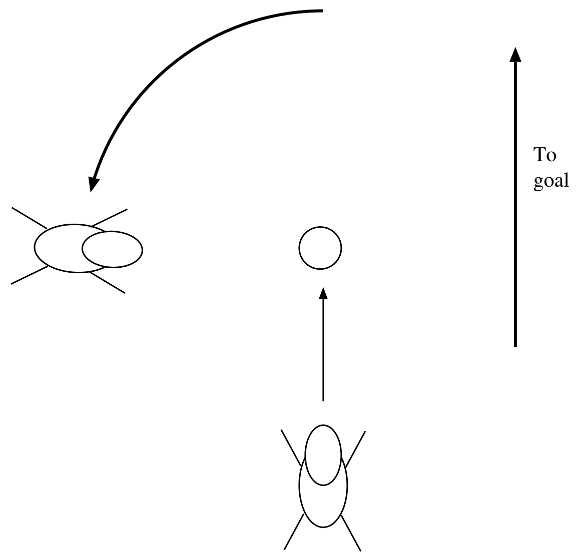


Figure 9.42: The standard strategy for ally avoidance. The robot heading away from the target goal circles around the ball to make way for the robot that is heading towards the target goal.

around. This approach proved simple and effective, as the robot would back off right out of the way, and with the rotation, it would often focus on another ball once it had finished backing away, giving way to its teammate. In addition, the head movement helped relocalise the robot.

In addition to the above, it was decided that the robots should only turn to ally avoidance behaviour at the early stages of its collection strategy. Clearly, if the robot is already turning with the ball to aim for the goal, it is much more efficient if the robot continues aiming if the teammate is nearby, and instead relies on its teammate to move out of the way. While risky, the risk added by this decision could be reduced by simply increasing the avoidance threshold, and the time saved proved to be worth the risk.

Another very simple idea that was tried was to mostly avoid the issue of ally avoidance by forcing the robots to head to opposite goals. While this ensures that they will stay clear of each other, since the balls are initially placed on one half of the field, one of the robots wastes a lot of time walking across the field to collect balls to shoot for the opposite goal.

Find Ball Home Positions

The find ball home position is the location on the field that the robots move towards in their find ball routines. This position effectively determines the field area that the robot will scan when searching for the ball. For the purposes of this challenge, with two robots and ten balls, these positions are crucial to minimising the chance that a ball may not be seen by either robot.

The initial configuration tried was to have both robots cover the half field in which the balls would be placed. While this seems reasonable, a problem with this configuration was that if the robots accidentally hit a ball reasonably deep into the other half of the field, they would not detect it when reverting to their find ball routines. The final configuration used avoided this problem by having one robot head towards the centre of the half in which the balls would be placed, and the other robot headed towards the centre of the field.

Recognising Multiple Balls

An optimisation strategy considered for the challenge was the possibility of recognising multiple balls. If the robots could detect multiple balls, they could then calculate an optimal order in which to collect the balls.

This idea was never implemented, as it would have introduced many complications in both the strategy and the world model. However, when separate orange blobs are seen by the robot, the vision system actually recognises the blob that is closest to the bottom of the image as being the ball. This means that theoretically, the robot would always recognise the closest ball, and thus always pick the closest ball to shoot.

Using Wireless Communication

In addition to the above possibility, wireless communication to coordinate the challenge was also considered, but ultimately not implemented. The primary advantage offered by wireless communication was coordination to ensure that the robots headed for different balls, and stayed out of each others way.

As with the possibility of recognising multiple balls, wireless communication was not implemented. The simple strategy of ally avoidance proved adequate for the purposes of this challenge.

9.5.3 Challege Result and Conclusion

UNSW ranked first in the ball collection challenge, shooting all ten balls in just over 2 minutes 18 seconds, well under the time limit, and was the only team to clear the challenge. The win secured UNSW's victory in the challenge competition overall, for the fourth consecutive year.

The results of the challenge are summarised below:

Team	Goals	Last goal time	Rank	Points
rUNSWift	**10**	**138.27**	**1**	**19**
CM-Pack'02	**8**	**161.68**	**2**	**18**
NUbots	**6**	**128.13**	**3**	**17**
RoboMutts++	4	179.3	4	16
German Team	2	37.9	5	15
Upennalizers	2	83.47	6	14
SPQR	2	85.27	7	13
ASURA	2	91.16	8	12
ARAIBO	1	16.34	9	11
McGill Red Dogs	1	60.20	10	10
UW Huskies	1	86.84	11	9
Wright Eagle	1	88.81	12	8
Borregos Salvajes	1	129.58	13	7
Baby Tigers	1	129.84	14	6
Les 3 Mousquitaires	0	-	15	3.5
Team Sweden	0	-	15	3.5
Essex Rovers	0	-	15	3.5
Cerberus	0	-	15	3.5
Yellow Jackets	Disqualified			

Chapter 10

Java Simulator

Section author: James Brooks and Nik Von Huben

10.1 Introduction

The RoboCup Simulator is a program which enables the user to simulate a game of soccer between two teams. The simulator models the playing field of the Sony Legged League in the RoboCup competition and the Sony robots themselves. The playing field environment consists of one ball, four robots (dogs) per side, one of each team being the goalie, and the field with a goal area. The dogs that are playing are able to interact with each other by running into each other, and they are able to interact with the ball via various kicks and just bumping into the ball.

A simulator was desired in the development of strategies for the Legged League Sony Robots for the following reasons:

- A simulator is helpful in the development of strategies because it allows the problem that we are trying to solved to be simplified. This simplification comes about in two ways. Firstly, the developer or programmer has less inputs into the strategy to deal with and they can also determine what inputs are supplied to the robots in the simulation. And Secondly, the actions performed by the robots are exact, whereas the electric motors on the real dogs are not exact, and thus the movement of the dogs is not exact.
- A simulator can be used to test many different types of strategies quickly because the user is not constrained by the physical properties of the dogs.
- Using the robots puts stress on their components, all of which have a limited life, especially in the heated battle of a game of Robot Soccer. The ability to test and develop strategies on a simulator reduces the amount

of time that the dogs are being used and thus increases the life of the components.

- The dogs themselves and the actual field are limited resources, that is, there are 9 robots, 1 regulation field, and 10 eager developers. Partitioning these resources is a hard task and the existence of a simulator for strategy development can be useful in alleviating the pressure in this area.

- In the particular case of RoboCup this year, there was a significant change in the rules between the 2001 competition and the 2002 competition.

These motivations will be expanded upon in the section “Goals and Motivation”.

10.2 Goals and Motivation

10.2.1 The Goal

Our goal in the development of this simulator for the RoboCup Legged League is to simulate the robots and the playing environment to a *sufficient* extent to allow meaningful and efficient strategy development to occur. It is important to understand the goal of the project because this goal has affected the direction taken in various areas of the project. Trying to simulate a robot, or anything in the real world, causes you to ask one question repeatedly, *"How much to I want to simulate?"*. The above goal simplified decision making concerning the extent to which the real robots should be simulated and reproduced.

10.2.2 The Motivation

Problem Simplification

The nature of the field of robotics, the fact that motors and cameras are not perfect, can hinder the development of strategies because these items introduce extra factors and thus extra complexities into the problem of developing a successful strategy. Having a simulator allows the strategy developer to develop a strategy without these extra inputs because the developer can test their strategy in an environment that can be perfectly managed.

An example in Robocup would be that the developer can supply the strategy (on a particular dog) with perfect data concerning it's position and where the other dogs are on the field. This method is similar to discovering heuristics for a given problem. The developer is able to relax some of the conditions or parameters of the problem to generate the strategy. In this way the developer may now much more easily be able to work out the components consistent in a good *overall* strategy.

Time Saving and Easiness

Testing strategies on the dogs takes up alot of time and can be frustrating. Playing matches requires batteries for the dogs. As batteries are used more and more, charging time increases and the usable time decreases. This was a particular nuisance towards the end of the development. Compiling and testing of the strategies in the Simulator is simply faster. There is no need to transfer the code to a memory stick to be placed in the dog. Dog boot-up time can be quite slow. Getting the dogs ready to play a game takes quite some time. With the whole testing procedure in the simulator many of these time-wasters and annoyances are removed.

Robot Wear

The robots themselves are built by Sony to lead a fairly relaxing lifestyle in the comfort of one's home. Playing soccer is actually quite detrimental to the joints of the robots, especially the neck motors and the clutch that is put in the neck joint to absorb any bumps to the neck. By writing a simulator that allows us to simulate large scale strategies we can eliminate a fair amount of wear-and-tear on the actual dogs themselves, thus increase the life of the components. Replacing the parts on the dogs can be a time consuming and expensive exercise.

Limited Resource Allocation

Thoroughly testing a strategy in a game environment requires the use of 8 dogs for certain periods of time (an official game lasts 20 minutes), although this can be shortened somewhat by various testing methods. Considering the fact that there were 10 members on the development team and that there were multiple areas in which development must occur, i.e Vision, World Model, Motion, and Strategy, the resources were quite limited.

Change in Rules

There was a change in the rules and in the field size. The field was increased in size by approximately 1.5 times in each dimension and an there was an additional dog to be in each team. Because of these large changes in the rules it increased the solution space of what strategies could be used dramatically. The simulator was to be developed to test out and possibly rate various types of strategies and give input into the development of strategies for the dogs.

10.3 Literature Survey

There are other simulators that exist also to simulate a game of soccer between two teams of autonomous robots.

10.3.1 The Robocup Soccer Simulator

The RoboCup Soccer Simulator [MCY02] program that is used for the simulation league is a research and educational tool for multi-agent systems and artificial intelligence. It enables for two teams of 11 simulated autonomous robotic players to play soccer. The soccer simulator is built around a system with various components. It contains a soccer server, a monitor and a logplayer.

The SoccerServer

The soccerserver contains the virtual soccer field environment which allows two simulation teams to play against each other, and it simulates all the movement of the ball and the players. It models many of the real world complexities that are involved in a real soccer game. The client programs connect to the soccer server to control a single player. Communication between clients/players is allowed through the soccerserver. This communication occurs on a peer-to-peer basis which is accurate and consistent with a real soccer match. It also contains a stamina model, goalies, and the handling of various other soccer rules such as offside. The soccer server contains a basic automatic referee which judges certain rules:

- Kick-Off. Making sure that players are in their own half.
- Goal. Updating the score,notifying all the players and resetting the ball.
- Out of Field. When the ball is out of the field.
- Player Clearance. Concerns where the players are allowed to be during different modes of play, such as kick-off, corner-kick etc.
- Play-Mode Control. Moving the game into play-ball mode when play is commenced
- Half-Time and Time-Up.

For rules which are more difficult to detect automatically a human referee is needed. These rules include:

- Surrounding the ball
- Blocking the goal with too many players
- Not bringing the ball into play after a certain number of cycles

- Intentionally blocking the movement of other players
- Abusing the goalie *catch* command, which allows safe movement of the ball anywhere in the penalty area.

The soccer server is a real time system but gameplay is calculated in discrete time units. All communication between the client and the server is done through UDP/IP packets. The server processes actions that come from the clients. These actions must be received during the time interval for the turn otherwise the player does nothing.

The SoccerMonitor

The soccermonitor displays the simulated game through the X Windows display system. In the display, information is available about the match in progress such as the score, team names and positions of the players and of the ball. Multiple monitors can be connected to the soccer server at one time (watching the same game). The monitor also allows the human referee to control the game. Even though interaction can take place between the human referee and the match through the monitor, a monitor is not actually needed for gameplay to occur.

The LogPlayer

The logplayer allows the playback of matches that have been played on the server in the past. When the logplayer is active on the server, it is able to save all the data of the match. This can be useful as it allows closer inspection of match-play to find weaknesses in strategy. As with normal playback software, you are able to navigate through the match file, to skip certain sections if necessary.

10.3.2 TeamBots: SoccerBots Simulator

SoccerBots is part of the TeamBots(tm) [TB00] software. TeamBots(tm) is a Java-based collection of application programs and Java packages for multi-agent mobile robotics research. The simulation environment is written entirely in Java. The TeamBots(tm) package allows robot control programs to be written in a simulation environment and then to be run on actual robots. TBHard allows TeamBots(tm) to run on the Nomadic Technologies' Nomad 150 robot.

TeamBots supports prototyping, simulation and execution of multirobot control systems. TeamBots also allows Robot control systems to be developed and these can be run in simulation using the TBSim simulation application, and on mobile robots using the TBHard robot execution environment.

SoccerBots simulates the dynamics and dimensions of a regulation RoboCup Small-Size Robot League game. Two teams of five robots compete on a ping-pong table by pushing and kicking an orange golf ball into the opponent's goal.

SoccerBots runs in the TBSim application using a configuration file that tailors it for soccer games. Other types of robots and tasks can also be simulated in TBSim, including the Nomadic Technologies' Nomad 150 robots Georgia Tech is using to compete in the AAAI-97 mobile robot contest. The same control systems that run in simulation can also operate real robot hardware using the TBHard application. At present, this capability is limited to Nomad 150s.

In the SoccerBots simulation the robots have various *sensors*, such as vectors to the ball, goal and other players, what player 'this' robot is, and the robot heading and field placing (among others), and *actuators*, such as kicking the ball, pushing the ball, or turning, and just standard movement, that they can use in the environment.

One of the restrictions that we found existed in the subsystems of this simulator was the fact that all objects on the field were simualted as circular objects. The ball is quite obviously circular in the real game and the robots in this league are also circular. Unfortunatley the Sony Legged Robots are not circular in shape. To employ circular shaped robots in the Legged League simulator would grossly change the accuracy of the simulator. It would not reflect the way that the actual dogs relate, especially in close interaction with each other, the ball and the walls.

10.4 Original Approach

To minimise the amount of work and time required to create the Simulator system, it was decided that the TeamBots software package was to be used as a basis from which the Robocup Legged League specific simulator system could be built upon. This initial approach was chosen since the TeamBots system was generic and structured well enough to allow for a separate system to be easily built on top of the existing code. After analysing the system further, a number of major issues were discovered which would make the development of the Simulator much more difficult then had been anticipated.

The first and most difficuly problem was that the generic robots in the TeamBots system were assumed to be circular rather then a user defined polygon shape. This assumption was also found in the collision detection algorithm which only calculated collisions between circles and hence did not support generic shapes as required by the Simulator. To continue implementing the Simulator using the TeamBots structure would require the construction of a entirely new collision detection algorithm and the supporting structure for generic shapes.

The second issue which made using the TeamBots system difficult was its complex, generic structure which made it hard to understand and follow. This complexity would have required excessive time to fully comprehend and so the decision was made to create a completely new structure which would allow future developers to improve the system easily.

These two problems led to the decision to create the Simulator system from the ground up rather then extend the existing TeamBots system.

10.5 Overview

10.5.1 The Game Environment

The game environment in the Simulator is very different to the real, physical environment the Robcup games are played in. Currently, the Simulator makes a number of assumptions in regard to the distribution of information which are not true in a real Robocup game. One of the major assumptions made by the Simulator is that all the players have global vision, which means that all strategies have precise knowledge of where all objects are on the field and which direction they are facing. Although this assumption seems far fetched, with the addition of wireless communication and increased accuracy in visual robot recognition, it could be quite possible to achieve this level of information sharing and accuracy. This topic will be discussed further under the section *Game Components*.

The second major assumption or generalisation in the Simulator is the shape of the dogs and the complexity of movement. The real Aibo robots have 4 legs, each with 3 degrees of freedom each of which are used continuously to propel the dog along the field. This level of complexity is not reflected in the Simulator and a generalisation is made so that the dogs are represented by a rectangular box approximately the size of a bounding box of an Aibo dog walking. This generalisation simplifies the movement and interaction of the dogs since there isn't independently moving parts and is explained in further depth in section 10.6.1.

Thirdly, the game environment in the Simulator differs to the real world in regard to kicking the ball. The Simulator doesn't calculate the physics of the action to determine the outcome and force produced on the ball, but rather uses probabilities and randomness to generate a similar effect as seen in a real Robocup match.

The final major difference between the Simulator and reality is that the Simulator works in a discrete time environment instead of a continuous time environment. Each discrete time step is split into four different phases : Update, Decision, Action, and Paint. The first phase is the Update phase where all the positional information for each of the objects is updated for that time step. The next phase is the Decision phase where each of the strategies playing in the match are permitted to make decisions and request actions. Next is the Action phase where all the actions requested by each of the strategies during the Decision phase are executed. Lastly is the Paint phase where the current state of the game is painted onto a viewable canvas and displayed in the GUI.

10.5.2 Objects

Basic objects

- Shape2D

- Vector2D
- Line2D
- SimpleObject
- Point2D

To help make the development of high level strategies easier, the Simulator contains a set of low level data structures which provide a collection of simple to use functions for commonly used objects. Each of these classes are also used throughout the Simulator system so that they can be used as a standard format for all strategy developers.

At the bottom of the hierarchy tree are the classes *Vector2D*, *Line2D* and *Point2D*. These classes are the basic geometric shapes used in the Simulator and come with a list of commonly used functions. All these classes are designed to be used in strategy development for greater code readability and easier mathematical programming.

The *Shape2D* class is an abstract class which represents a two dimensional polygon. All the objects on the field require an instance of a class derived from *Shape2D* so that collisions can be detected and so that they can be drawn. In the Simulator, the *DogShape* class and the *CircleShape* class are both derived from *Shape2D* and are used as the physical and visible representation of the dogs and the ball respectively.

The *SimpleObject* class is one of the most general classes in the Simulator and is the class from which all the physical objects in the game are derived from. The class contains variables which relate to physical attributes such as mass, friction and shape, and is only responsible for drawing itself.

The Controller

- Controller

The Controller class, as the name implies, controls the running of a match in the Simulator and is responsible for the construction, destruction and management of all the classes relevant to playing a match. The objects which the Controller creates and manages are the ball of class *Ball*, the playing field of class *PlayingField*, all the strategies on the field, and the action manager which is of the class *ActionManager*. The Controller class is also the referee of a match and is responsible for counting goals, enforcing game rules and keeping track of game statistics. It is also responsible for controlling the timing of the game which controls how much 'time' there is between each calculation step, how often strategies receive updated information and how often strategies can make decisions. Lastly, the Controller is responsible for distributing the information in a game which affects the how each strategy sees the world, and for drawing the game onto a canvas.

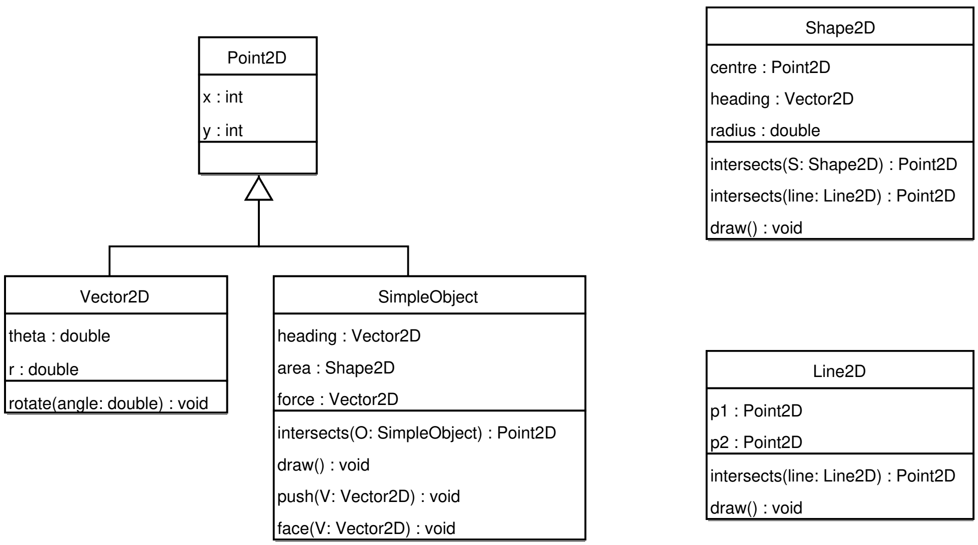


Figure 10.1: UML diagram of the basic classes in the Simulator.

The ActionManager and Action

- ActionManager
- Action

The *ActionManager* class is the engine of the Simulator and is responsible for calculating and controlling the movement of all the objects on the field. In essence, the *ActionManager* is a collision detection engine which calculates if two objects hit each other and what the corresponding effects are. In the Simulator, there is only ever one instance of the *ActionManager* class which is used by the *Controller* class to calculate the movement of all the objects in the game. To have greater control over the Simulator, all actions which affect other objects on the field are restricted so that they must go through an action manager, that is, only the *ActionManager* class can perform actions. For a strategy to perform an action, it must make a request to the action manager asking it to perform that action on its behalf. This is done through the *Action* class.

The *Action* class is a basic structure which defines how an object can alter itself and affect others in a match. An action is not allowed to perform tasks which can affect other objects, but must call functions in the *ActionManager* object to perform tasks on its behalf. It is however allowed to perform taks which alter strategy dependent variables without the need to call functions in the action manager.

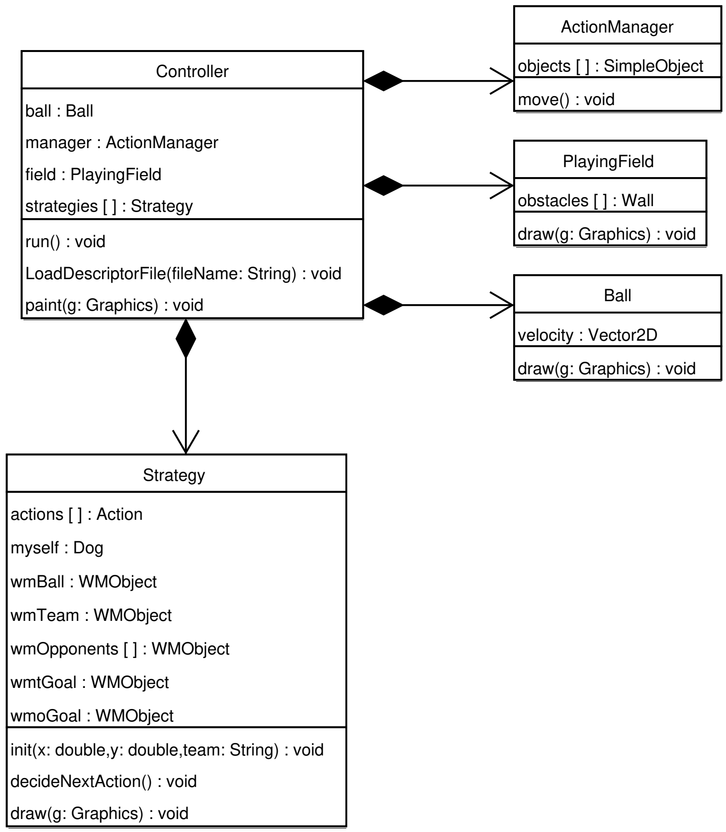


Figure 10.2: UML diagram of the Controller and associated classes.

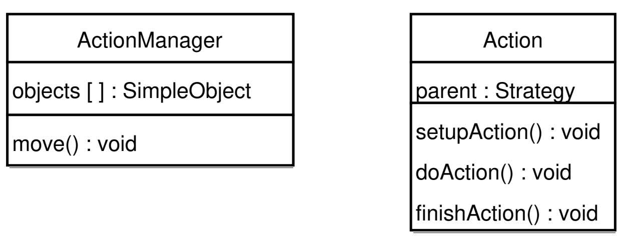


Figure 10.3: UML digram of the Action and ActionManager classes.

The Dogs and Strategy

- DogShape
- Dog
- Strategy
- WMObject
- Team

The classes related to strategy development are divided into two separate areas: one relates to the representation of the physical dog in the Simulator and the other relates to the strategy and its perspective of the physical world.

The *Dog* class represents a physical Aibo robotic dog in the Simulator and contains the actual values for the Aibo's dimensions, the heading and position of the dog and other related constants such as mass. The *Dog* class also contains an instance of the *DogShape* class which is a *Shape2D* representation of the physical dog. This *DogShape* object is used by the action manager to calculate collisions with other objects on the playing field.

The *Strategy* class is the basic structure from which all strategies are derived and is the decision making component of a robot in a match. A strategy interacts with the world by activating the various actions it contains, and receives information from the world through its local world model which is updated by the Controller. The local world model is a local copy of the vital information from each of the objects in a match, as viewed from the perspective of a strategy. This information is stored in *WMObject* class objects and is updated by the Controller which can add varying degrees of noise. A *WMObject* class object represents a world model object and is derived from the *SimpleObject* class.

To make the loading of multiple strategies easier, the *Team* class is provided which is responsible for creating multiple strategies at once and for initialising each strategy's starting values. These strategies, once created, are passed to the Controller which keeps a reference to them so that they can be used in a game.

Miscellaneous game play classes

- Ball
- CircleShape
- Wall
- PlayingField
- Simulator

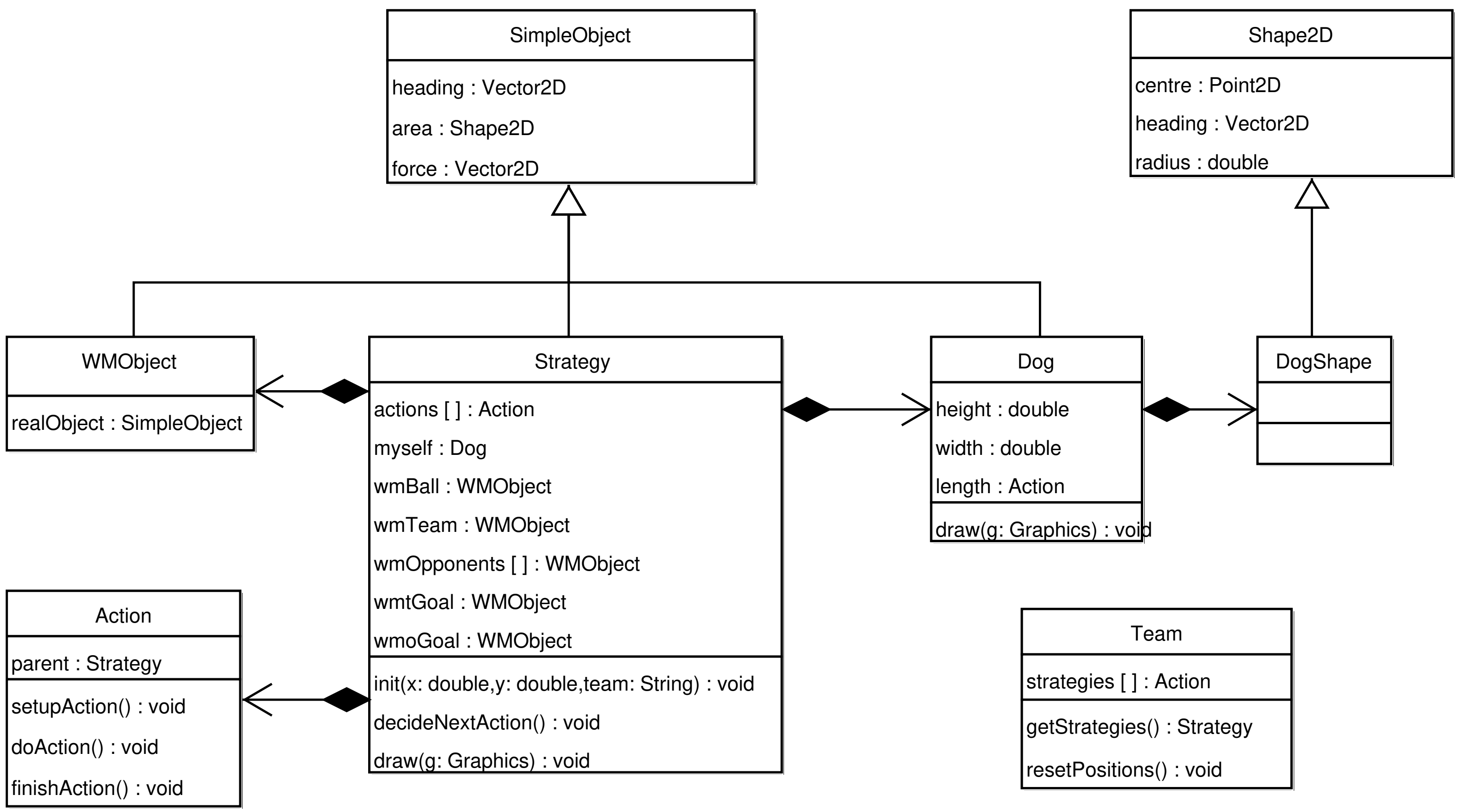


Figure 10.4: UML diagram of the class associated with a strategy.

This group of classes make up the final components required to play a match in the Simulator.

The *Ball* class is derived from *SimpleObject* and represents the ball in the match. Its responsibilities are to draw itself and to provide a shape which interacts with other objects on the field. The *Ball* class contains information of its physical size and weight, and also contains an instance of the *CircleShape* class as its physical representaion. The *CircleShape* calss is derived from *Shape2D* and is an implementation of a circular polygon.

Similar to the *Ball* class is the *Wall* class which is also derived from *SimpleObject* and represents a wall or physical, immobile barrier. Its responsibilities are the same as those for a ball, the only difference being the *Wall* class has a line as its shape and does not move.

The *PlayingField* class is a structure used to construct the officially sized Robocup playing field used in a match. It creates the appropriate walls in the correct position and is also responsible for drawing the field and walls.

The *Simulator* class is responsible for the interacting with the user via a GUI and initialising the entire Simulator system. It is also responsible for creating the GUI and the Controller and linking the Controller object to the viewable canvas. The *Simulator* class also responds to user input and passes the appropriate information to the *Controller* class.

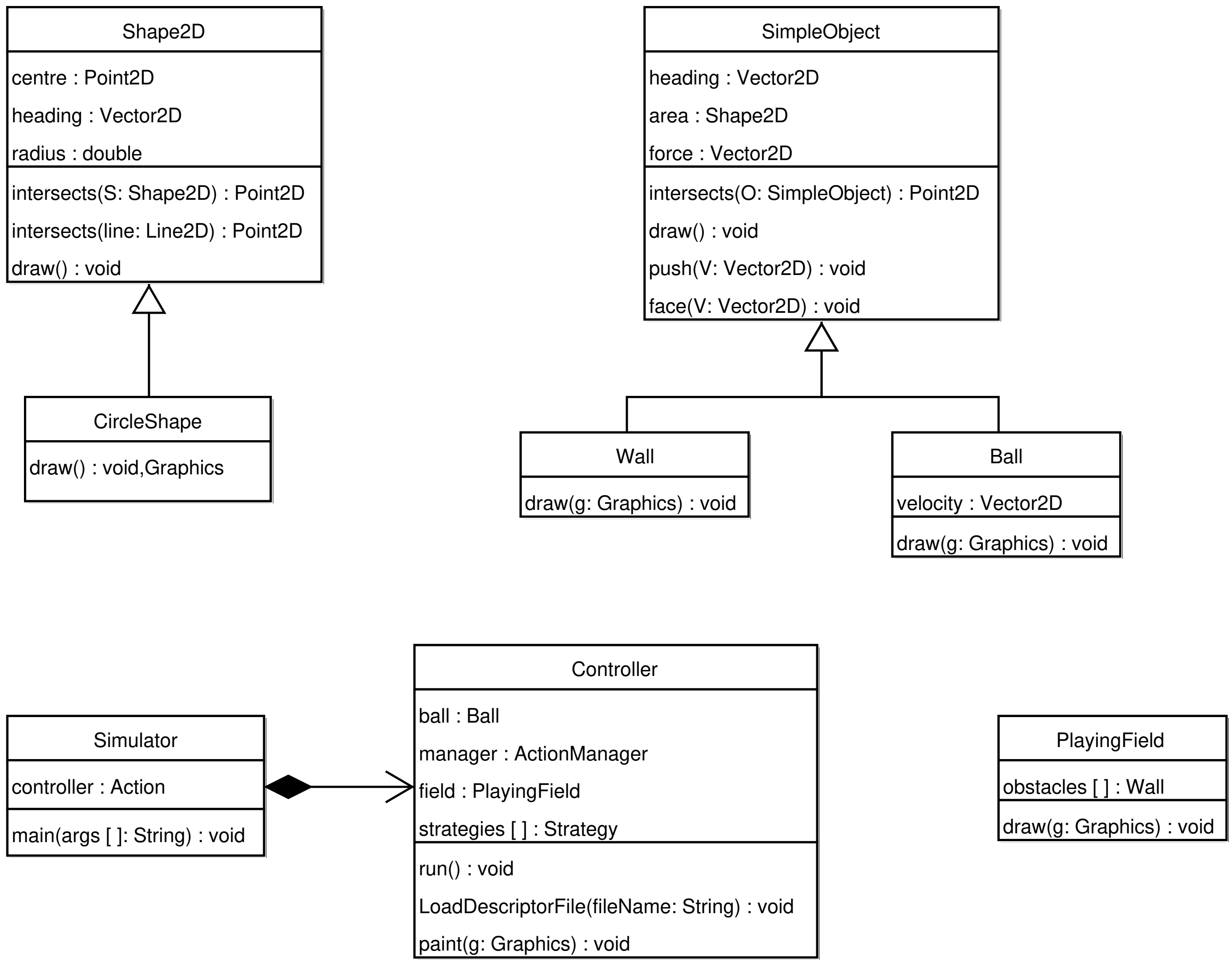


Figure 10.5: UML diagram of the miscellaneous game play classes.

10.6 Game Components

To simulate a complete Robocup Legged League match accurately it is important to identify the main components of a game. Firstly, the robots on the field must be able to move around the field freely and be able to interact with each other and with the ball. These complex interactions as a result of motion are often key turning points in a game and force strategy developers to constantly invent increasingly accurate actions for the robots to perform.

The second component of a Robocup match is information and how it is received through the various sensors. In a Robocup strategy the information gathered from the various sensors are combined together into a world model which represents the current state of the game from the perspective of that particular strategy. Each of the strategies has their own unique world model based upon their own view of the world and upon information transmitted wirelessly from other team mates. This world model is the basis from which a strategy makes its decision and is hence also a major component in the Simulator.

The third and final component of a Robocup match is the actual strategy or decision making algorithm. This component is the most flexible part of the Simulator and differs to the other components by being completely user defined having no limits or boundaries.

These three components of the Simulator system, motion, sensors and strategy, will be expanded upon in the following sections.

10.6.1 Motion

Dog

To create the effect of continuous motion found in a Robocup game, the Simulator calculates the movement of each of the dogs in small discrete time steps. The length of these time steps can be changed by the user so that the Simulator can run with increased or decreased accuracy at the expense of computation time. The actual motion of the dogs in the Simulator is controlled by a number of variables and data structures spread throughout the Simulator to allow for easy development and improvement. The classes involved in moving a dog around the field are the *Action*, *ActionManager*, *Controller* and *Strategy* classes.

To understand how the Simulator creates the effect of motion it is helpful to see the flow of information through the Simulator structure when a strategy makes a decision to move. Lets pretend we have a scenario in which a strategy, that is, a class which is derived from the *Strategy* class, has decided to move forwards and turn slightly during the Decision phase of the Simulator. To make this movement happen, the strategy must set the *forward*, *left* and *turnCCW* variables to the values desired and activate the Walk action. This can done through a single call to the makeParaWalk function giving the *forward*, *left* and *turnCCW* values as arguments. After these values have been set, the strategy must wait until the Action phase of the Simulator for the movement to occur. In the Action phase, the Controller inspects every action of the strategy and allows each activated action to perform its task. The action which has been activated, in this case the Walk action, uses the *ActionManager* object to perform tasks which can affect other objects in the game, such as movement. This is done by the action calling the *move* function in the action manager and passing it the desired movement vector, the angle of turning and its parent's *Dog* object as the *SimpleObject* object to act upon. The action manager then calculates the movement and checks for any collisions before moving the *SimpleObject* object to its new position. This then completes the sequence of events which takes place to move a strategy.

Another frequently occuring scenario is when a strategy decides to kick the ball using one of the available kcking actions. In the same fashion as in the previous scenario, the strategy must activate the appropriate action and assign the variables required by that action to the desired values. This decision takes place in the Decision phase of the Simulator, and it is not until the Action phase that the activated action is performed. For a kicking action, the action must call the *kick* function in the action manager which calculates whether or not the dog is close enough to the ball to act upon it and if that is the case, pushes the ball in the desired direction with the desired force.

This sequence of decision and action happens every time step for every strategy in the game so that the motion of all the dogs appears to be smooth and continuous.

Ball

During a game in the Simulator, the ball is in constant motion due to the forces placed upon it by the other objects in the game. The ball never acts upon itself nor makes any decisions, but only responds to the effects caused by interactions with the dogs on the field and the walls which bound it. Due to this unique nature of motion, the balls movement is calculated in a slightly different manner then that of the dogs. The ball does not have the *forward*, *left* and *turnCCW* variables as the strategies have, but rather has a vector called *velocity* which represents the speed and direction the ball is moving in centimeters per second. Each time step in the Simulator the ball is moved by the Controller in the Action phase by using the ActionManager *move* function with the balls velocity vector used as the direction of motion. After the ball has been moved, the Controller reduces the speed of the ball by using the following equation.

$$V_{new} = V_{old} * ball\_friction^{1/fps}$$

The equation is designed so that it reduces the speed of the ball by a constant percentage every Simulator second. This constant percentage is stored in the ball variable called *friction.*

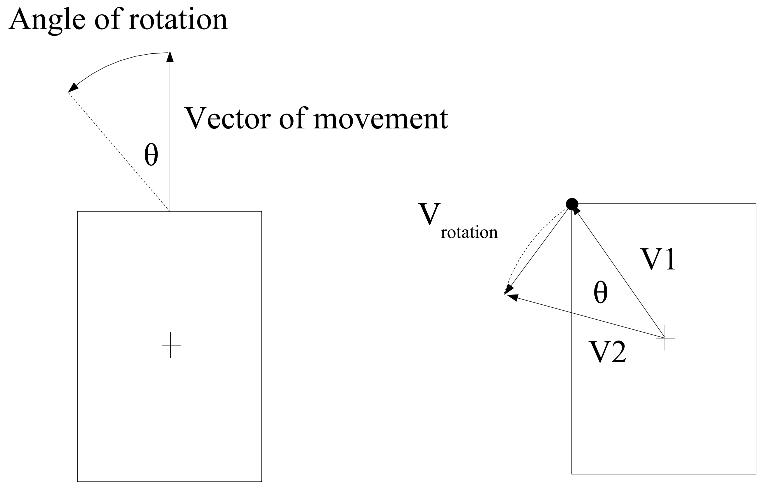


Figure 10.6: The construction of the *vector of rotation* of a point using the *vector of movement* and *angle of rotation*.

Collision Detection and Correction

An intergral part of all simulators which have objects in motion is that of collision detection and collision correction. Without these two components the Simulator could not simulate a game of Robocup soccer as both the dogs and the ball would be moving in and out of each other and the field. Both collision detection and correction are calculated whenever any object in the game moves, and this is done in the *ActionManager* class. For the action manager to determine whether or not two objects will hit each other as a result of movement, both objects must have an instance of a class derived from *Shape2D*. In the case of a strategy, the instance is of a class called *DogShape* and it represents the physical shape of the dog, whilst in the case of the ball, the instance is of the *CircleShape* class which is simply a generic class derived from *Shape2D* for defining circles. Using any two shapes, the vector of movement and the angle of rotation, the action manager performs the collision detection using a technique called movement line collision detection.

The movement line collision detection technique can determine not only if two objects intersect as a result of movement, but can also determine the exact point of collision and the vector of movement at that point. This is achieved by constructing the movement lines of the moving objects shape and calculating the nearest point of contact with the second objects shape. The movement lines of a particular a shape are calculated using the vector of movement, the angle of rotation, and the points which define the shape. These lines define the path of movement of each point of the shape.

To construct the movement line of a point on the shape, a vector is constructed from the centre of the shape to one of the points which define the shape as shown in Figure 10.6.

That vector, V_1, is then rotated by the angle of rotation to form a new vector

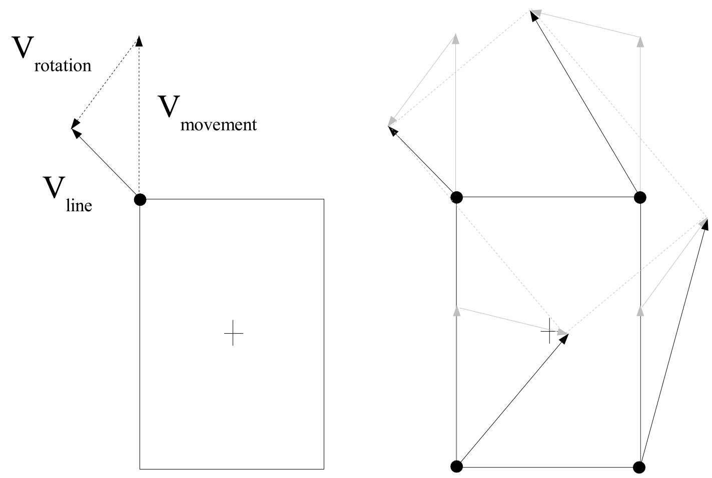


Figure 10.7: The movement lines for each point in a shape.

called V_2. The vector of rotation for that particular point is then given by the equation $V_{rotation} = V_2 - V_1$. This vector of rotation is added to the vector of movement to form the line vector for that point. The line vector is then used to construct the movement line by creating a line which starts at the selected shape point and ends at the shape point translated by the line vector.

This procedure is repeated for each point so that the movement line for each point is constructed as shown in Figure 10.7.

Once the movement lines for the moving shape have been constructed, the collision detection can commence by calculating the line intersections between the lines which construct the second shape and the newly constructed movement lines. In Figure 10.8 it can be seen that Shape 1 will collide with Shape 2 as a result of the movement and this is detected by the intersection of one of the movement lines with one of the shape lines.

Though the collision is detected in the case above, only calculating the movement lines for the moving shape does not detect every collision. Consider the case as shown in Figure 10.9.

In this case, Shape 1 does collide with Shape 2 but because the movement lines do not intersect with any of the lines of Shape 2, the collision is not detected. To overcome this problem, the situation is examined in reverse by considering the movement of Shape 2 as seen by Shape 1. Instead of Shape 1 moving towards Shape 2, it is equivalent to consider Shape 2 moving the reverse direction towards Shape 1. Similarly with rotation, instead of Shape 1 rotating counter clockwise on its centre it is equivalent to consider Shape 2 rotating clockwise about the centre of Shape 1. After the movement lines for Shape 2 have been constructed the line intersection test will detect any collisions as shown in Figure 10.10.

If an object does not collide with any other object as a result of a movement, the action manager allows the movement to happen and updates that objects position and angle of rotation. However, if a moving object does hit another ob-

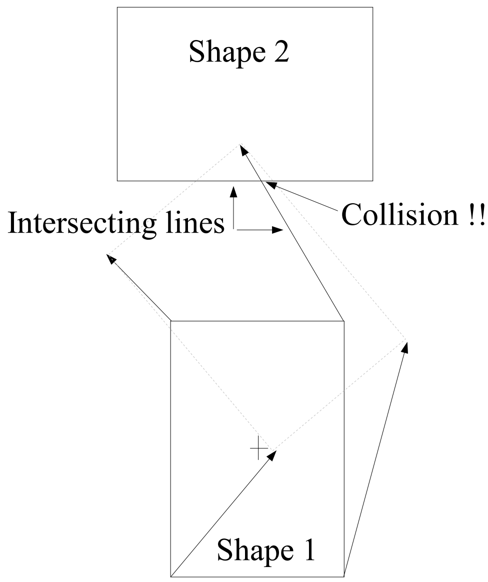


Figure 10.8: How movement lines are used to detect collisions between two shapes.

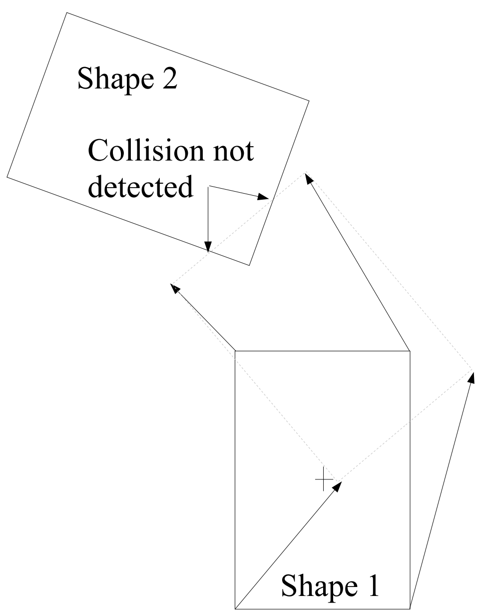


Figure 10.9: An example where the movement lines from only one of the two shapes cannot detect a collision.

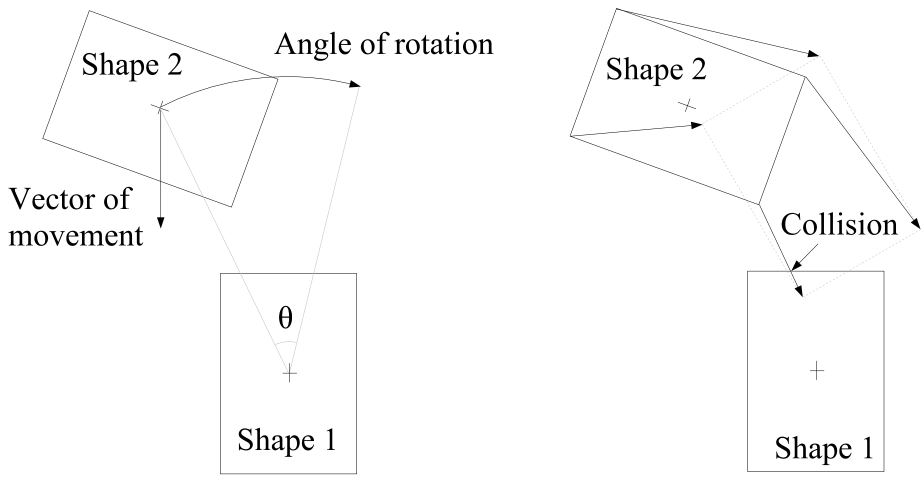


Figure 10.10: The construction of the movement lines of the stationary object (Shape 2) as seen by the moving object (Shape 1).

ject the action manager does not allow the movement to happen and the moving object remains stationary, forfeiting its turn to move. This simple approach to collision correction can cause problems in the Simulator when two objects are both moving into each other, resulting in a jam where neither of the objects move and become stationary for the entire length of the game. This problem is overcome by the addition of a simplified concept of forces which allows colliding objects to affect one anothers movement.

Whenever two objects collide, the vector of movement at the point of contact is calculated and used to represent the size and direction of the force exerted by the moving object onto the second object. This force is stored in the *Vector2D* variable called *force* which is found in the *SimpleObject* and derived classes and represents the size and direction of the force being exerted on that object at a particular moment in time. This vector is altered by calling the *push* function of the *Simpleobject* object that is experiencing a force and passing the pushing vector as an argument. The *push* function updates that objects force vector in relation to the length of the given vector, the mass of the object and the size and direction of the existing force vector. As well as pushing the object it collides with, the moving object is also pushed as a result of the collision. The size and direction of this reverse pushing vector is related to the angle at which the objects collided so that the moving object is pushing perpendicularly away from the second object. This perpendicular force is simply the component of the pushing vector which is perpendicular to the object's shape at the point of intersection.

The force vector is only updated and used by the action manager and only when an object requests a movement or when an object collides with another object as a result of movement. When an object requests to be moved in a particular direction, the action manager adds that objects force vector with the given movement vector to create the final vector of movement. After the force vector has been used to calculate the final vector of movement it is then reset to length zero to ensure that the forces created by collisions do not accumulate.

10.6.2 Sensors

Due to the unique game environment of the Simulator, the concept of sensors is inapplicable and would not be beneficial to the system since every strategy is assumed to have a complete and perfect perspective of the game environment. Instead of having a collection of sensors to allow a strategy to perceive its environment, each strategy has a world model which represents the entire game environment at a particular point in time as seen from the perspective of that strategy. Currently, each strategy's world model is a perfect representation of the game environment and hence does not contain any noisy or false data.

World Model

The world model is a collection of *WMObject* class variables, one for each object on the playing field, and is found in the *Strategy* class. Each *WMObject* variable contains the positional and directional information of a particular object as seen from the perspective of that strategy and is updated every time step by the Controller during the Update phase. Some of the world model variables are grouped into sets such as the *wmTeam* and *wmOpponents* arrays which contain the world model information on all the dogs on the field from both teams. The only other objects which are contained in the world model are the ball and the two goals which are stored in the *wmBall*, *wmtGoal* and *wmoGoal* variables respectively.

To update the world model of a particular strategy, the Controller calls the *updateStrategyWM* function giving the strategy as an argument. The *updateStrategyWM* function updates a particular strategy's world model variables with the current real values from the Controller, translating and reflecting them as appropriate for the perspective of that strategy. This is done during the Update phase of the Simulator.

10.6.3 Strategy

Development Environment

The *development environment* consists of three parts:

- What data is available for the strategy to use.
- How the data is made available for use.
- How the strategy invokes actions on the 'dog'.

Similarity with Actual Development Environment

The aim was to have the environment for strategy development in the Simulator similar to the development environment on the dogs. This was desired because it would be easier to transfer the the strategy from the Simulator code into the strategy code on the actual dogs. When more complex strategies are developed on the Simulator this could be extremely useful as the work required to modify the code would be greatly minimised.

Another reason for making the development environment similar is ease of use for the developers. It is very likely that the strategy developers will be familiar with the development environment on the dogs. Because of this most of the knowledge that is required in order to write a strategy is already known and there are only a few points which need to be observed on the Simulator that are different from the dogs.

- wanted to have the programming environment for the strategy as the same as the environment on the dogs - we wanted this because then it would be easier to transport the strategy from the simulator to the dogs

- also because the actual code environment would be familiar which would just make writing the strategy simpler because you do not need to learn another environment.

The Simulator Environment

When writing a strategy for the Simulator the first function that is needed is the **DecideNextAction** function. This function, like on the real dogs, is called every 'cycle' and the strategy can make decisions, independent of each other, every cycle. The decision making and acting is done within this function. Now within this function the strategy can access a number of variables which it can use in order to make it's decisions. These are the sensor variables and they provide information on where the dog is, and where other objects are in the game.

The variables available are:

x, y and heading represent the x,y position of the dog, and which way the dog is facing on the field. The *heading* variable is a vector.

wmBall is a *WMObject* which represents the ball. It contains information that relates to the ball which is accessible by the dog, that is, it's position, x and y, and the heading of the ball from *this* dog. These variables are part of the *WMObject* class.

wmtGoal is a *WMObject* which represents the target goal. It contains information on the position of the target goal and it's heading in relation to *this* dog.

wmoGoal is the same as the *wmtGoal* but it represents the dog's own goal.

wmTeam is an array of *WMObject*'s representing the teammate dogs. It contains the standard information on them such as their x and y position and the heading to them in relation to *this* dog.

wmOpponents is the same as *wmTeam*, but it is an array of the opponent dogs.

These variables provide all the information needed to decide what actions need to be taken.

To make the dogs move and do different actions these actions must be activated. They can activated by the following code fragment:

makeParaWalk(WALK, Forward, Left, turnCCW);

The actions are contained in the *actions* array and can be accessed via a number of constants. The actions available and their respective constants are:

FACE_THE_BALL sets the dogs' turnCCW such that the dog will turn towards the ball.

WALK allows the dogs' *Forward*, *Left* and *turnCCW* variables to be accepted and used for he future walking action.

BASIC_KICK is a standard kick where the ball is propelled forward.

FRONT_KICK_W_DEFLECTION is similar to *BASIC_KICK* but instead of the ball just going forward, the ball can be given a slight deviation within 15 degrees.

TURN_KICK is a kick in which the dog 'grabs' the ball and turns with it. After a certain amount of turning the ball will shoot out.

The *forward*, *left* and *turnCCW* variables control the movement of the dog's. *Forward* and *left* are measured in centimetres per second and *turnCCW* is measured in degrees. These variables must be supplied to the *makeParaWalk* function which will propagate the changes.

10.7 Testing and Results

To test the accuracy of the Simulator and to get some preliminary results in comparing strategies, three different strategies were created and played against each other. This testing confirmed a theory used to create Robocup strategies and revealed a number areas within the Simulator system which required improvement.

10.7.1 Strategies

KickDog

The first and most simple strategy created was called KickDog which simply chased the ball, circling it to get behind it, with the aim of kicking it towards the opponent goal. From this strategy a KickDog team was created which had three identical KickDog strategies as the forward players and a standard goalie strategy as the goalie.

Advanced KickDog

Improving from the KickDog strategy was the Advanced KickDog strategy which implemented a dynamic role assignment system which each strategy used to determine if it was a defender, an attacker or a backup attacker. Three out of the four strategies in the Advanced KickDog team use the Advanced KickDog strategy which dynamically assigns itself to one of the three roles every time it makes a decision. This decision is made by using the following rules:

- If I am closest to the ball, I am the attacker.
- If I am furthest away from the ball, I am the defender.
- If I am neither the closest nor the furthest away from the ball, I am the backup attacker.

The attacker strategy behaves exactly like the standard KickDog strategy by chasing the ball attempting to kick it towards the opponent goal. The backup attacker strategy also behaves like the standard KickDog strategy, but has a condition which prevents it from going closer then 40 cm to the ball, whilst the defender strategy constantly positions itself half way between the ball and its own goal.

Region Player

The Region Player is a team in which the players are based around regions in the field. There are three regions in this version of the region player:

- the forward half of the field, Region 1.
- the back half of the field, Region 3.
- the region that runs from the forward quarter line to the back quarter line, Region 2.

The players are effectively comfined to these regions. When a dog is outside it's region it aims to move back into it while also taking a route in which closest to the ball. The players in each of the regions involve different minor strategies which then combines into the overall strategy for the whole team. This means that the dogs try to work in co-operation and that they are spread out over the field.

One central part to this strategy is how it works out the best way to kick to a particular point. For instance if a dog wants to kick to a particular point on the field but there is a dog in the way, then it can shoot either side of the dog. The best way to kick can then be approximated by how much *deviation* each kick has from the desired kick direction. The direction with the least deviation will be the best way to kick. This function is used commonly between the dogs which is very helpful because it allows teammate dogs to *predict* which way the teammate with the ball will kick, and thus the teammates can place themselves in a good position in which the teammate with the ball is likely to kick.

The Region 1 player minor strategy is to score. It is the dog which is always in the opponents half, waiting for the ball to be kicked up by it's teammates. The strategy consists of two main parts. If the dog is the closest to the ball then it will either shoot for goal, or pass. This will be decided by assessing how successful each option is likely to be using the 'best way to kick' method described above. If the dog is not the closest to the ball it will then move into a 'good' receiving position for a pass which is again calculated using the 'best way to kick' method.

The Region 2 player minor strategy is to pass the ball up the field or to score. It acts very similar to the Region 1 player but it's region is in a different location. The different region location causes this player to pass the ball to the Region 1 player more often than it would choose to score because it is likely that the path to the goal is more blocked by opponent players than the path to the Region 1 teammate.

The Region 3 player minor strategy is just to kick the ball up the field. It doesnt take into account anything concerning scoring or even ball placement post-kick. As a consequence, the Region 3 player is simpler than the other region players. If the ball is in region 3 then the Region 3 player will attack the ball and try to kick it in a direction which is up the field. This direction is worked out by assuming the dog wants to kick to the target goal then by using the 'best way to kick' method it works out the direction in which it should kick. If the ball is not in Region 3 then the dog remains in defence mode where is lines itself up with the ball and the goal.

10.7.2 Results and Conclusions

Strategy Analysis

Since each strategy implemented different approaches to the problem, they each had their own unique set of advantages and disadvantages. The KickDog strategy, for example, often swamped the ball with all three forward players preventing the opposition from clearing or passing the ball. This method worked well when all of the KickDog strategies were close to the ball, but it was also a disadvantage when the opposition cleared the ball away from the group. Since all three forward players were constantly chasing the ball, they were often grouped together close to the opponents goal leaving their own half undefended. Whenever the opposing strategies kicked the ball into the KickDog teams half, there was only the goalie strategy there to defend.

The Advanced KickDog strategy improved greatly from the standard KickDog strategy eliminating the grouping behaviour of the forward players and ensured that there was always a player defending in its own half. There were no other advantages in attacking over the KickDog strategy, only an increase in defense.

The best strategy out of the three, the RegionPlayer strategy, had a key advantage over the other two strategies by having the three forward players evenly spaced across the field. This ensured that at almost any point in the game there was always a RegionPlayer strategy near the ball, either in attack or in defence. It also prevented the opposing players from having control over the ball for an extended period of time, limiting their opportunities for attack. The method had a disadvantage, however, which was that individual RegionPlayer strategies could be easily swamped by a group of attackers and would often lose the ball.

After analysing each of the three strategies, a single concept was extracted which seemed to summarise the best approach to playing a game of Robocup soccer. Simply put, the Simulator supported the idea that constant movement of the ball was essential to a successful game of soccer. If a strategy took too long to move the ball to a new location it was invariably swamped by opposing players often leading to a lost ball or a failure to take advantage of the situation. This concept sounds simple, but is important to note that often speed rather then accuracy often won at the end of the day.

Simulator Analysis

Overall, the Simulator successfully simulated the Robocup Legged League soccer environment, although after running the Simulator through the series of tests, a number of irregularities were identified which reduced the realism of the Simulator and created some minor problems. These irregularities are outlined below and some possible solutions are discussed further in Section 10.8.

The first irregularity detected was in the motion of the robots as they seemed to be moving and responding to the motion of the ball too quickly. Though this takes away from the realism of the Simulator, it shouldn't reduce the effectiveness of it since all the strategies move at equal speeds, and the Simulators aim being strategy comparison rather then simulation perfection.

Also related to the motion of the strategies is the occasional problem of robot jamming where two or more robots become jammed against each other or against the wall. On most occasions, the robots manage to free themselves after a short period of time but when that doesn't happen all the strategies in the Simulator need to be reset back to their kick off positions. This problem reduces the realism and effectiveness of the Simulator significantly especially since events like this rarely happen in the Robocup competition.

The third irregularity occurs when a strategy kicks the ball whilst the ball is already in motion. Instead of ball continuing on its course whilst the strategy is performing that action, the ball stops moving with its original velocity and starts moving with the velocity as given by the kick. This creates the illusion of the robot catching the ball, particularly when the strategy is performing the turn kick action.

10.8 Future Improvements

The potential size and complexity of the Simulator goes far beyond its current state of functionality. There are many possible improvements of varying size which will make the Simulator a more accurate representation of the real Robocup Legged League competition. Since the structure of the Simulator can be broken up into three different game components, the possible future improvements have also been divided into those same categories.

10.8.1 Motion

One of the major generalisations made by the Simulator is in regard to the shape used to represent an Aibo robotic dog. Currently, the shape used by the Simulator has been generalised to a rectangular box which bounds a Aibo robotic dog when it is standing stationary in the ready mode. This shape is used by the Simulator to detect collisions between objects on the field. By improving the accuracy and imcreasing the complexity of the shape used to represent an Aibo dog, the Simulator would be able to simulate collisions much more realistically and would make the Simulator a more accurate strategy development tool. Some practical ways in which this can be achieved are by adding other body parts to the shape such as legs or a head. These additions may be made just to the shape itself, or may be considered to be complety separate shapes to the body allowing each part to have its own separate movement parameters. The latter suggestion would allow each leg shape to move forth and back in a periodic movement in a similar fashion as the real Aibo dogs. This will not only make the Simulator look more realistic, but will also improve the realism in regard to ball interaction since each leg will be able to interact with the ball.

Another improvement which will increase the accuracy of the Simulator is to improve the collision detection and collision correction algorithms found in the *ActionManager* class. A major problem in the Simulator is that two or more dogs can often become jammed against a wall. When this happens, the user must reset the Simulator and move the dogs back to their kick off positions so that the game can continue. One possible solution to this problem is create a realistic physics engine which would calculate the forces, velocity and movement of all the objects using the appropriate physics equations. This solution would also solve other minor problems inherent in the current action manager implementation and would simplify the addition of further improvements such as angled walls.

10.8.2 Sensors

As mentioned earlier in Section 10.6.2, the Simulator does not support the addition of extra sensors such as vision or wireless communication. These sensors are a vital part of game play in the Robocup Legged League competition and would greatly increase the accuracy of the Simulator if they were added. Currently, the Simulator system does not contain any class structures which would support the addition of these sensors and so a redesign of the system would be required. Adding this structure to the system would allow for easy implementation of extra sensors such as vision or wireless communication and would improve the structure of the existing world model sensor.

Another improvement to the sensor sub-system would be the addition of realistic noise to the information available to the strategies. This improvement would also be enhanced by the redesign of the sensor sub-system by allowing each individual sensor to control the amount of noise at a particular point in time.

10.8.3 Strategy

A minor improvement which would assist in strategy development is to increase the amount of information displayed in the graphical user interface during a game. Information such as a strategy position, its current action or other native variables will assist developers in debugging a strategy's algorithm.

Chapter 11

Wireless

Section author: Zheng David Wang

11.1 Introduction

In this year's Sony Legged League, Wireless Communication is introduced into the competition that may well revolutionise the way the game is played. It allows robots with an AIBO Wireless LAN Card (PCMCIA) to communicate between each other. This is like having psychic abilities in a real soccer game. The wireless network runs on the IEEE 802.11b network protocol with a bandwidth of 11 Mbps. This is much faster than the COM port communication tool that we used to have for debugging robots which only pumps data at a miserable 115.2 Kbps. This opens up endless possibilities. Debugging can be made much easier with a tool that allows real time error message printing during a match, see what the robot is thinking, display the robots vision or its world model. Plus, no wire no mess. Knowledge sharing among team-mates is another possibility, e.g. ball location, team-mate location, opponent location. In theory a robot can play blind folded by only using the information from its team-mate. Tactic planning (collaboration) is achievable, e.g. ball passing, better opponent avoidance, backing off from team-mate. Of cause we can also have a little fun with real time human control, e.g AIBO vs Human match. In fact Sony has ran a Celebrity vs AIBO in Japan during the RoboCup Competition. Unfortunately, their game controller is very slow, making the humans terrible players. If they used our game control, the tied would have been turned.

11.1.1 Equipment

To conduct wireless communication the following list of items are needed.

1. AIBO Wireless Card. Inserted in the PCMCIA slot in the AIBO core.

2. 802.11b compatible wireless card or hub attached to the base station computer.

3. Aperios Wireless Library for establishment of the base station computer.

11.2 Wireless Communication Infrastructure

Communication between the robots are conducted using IEEE 802.11b and TCP/IP. In order to adhere to Sony's strict Subject-Observer pattern which they have uniformly applied through out its operating system –Aperios, they have created an object called TCPGateway that will manage the data transfer via TCP/IP. In other words, programmers will not be exposed to the standard c++ TCP/IP libraries, rather they have to use Sony's objects for communication. To develop an understanding for the infrastructure, I will first present a brief outline of the Subject-Observer pattern.

11.2.1 Subject Observer Pattern

This defines a many-to-many dependency between objects so that when the *subject* changes state all the *observer* will be notified and updated automatically [EG95]. A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain their consistency between objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces reusability. Consider a program that provides different pie chart and bar chart displays for a data set. You would want to be able to reuse the display objects for other data and reuse the data object to allow it to be displayed on any chart and update the display at the same time when the data changes.

The key to this pattern are *subject* and *observer*. An *subject* may have any number of dependent *observers*. They can be varied at runtime. A *subject* does not need to know how many *observers* are interested in its data. All *observers* are notified whenever the *subject* undergoes change. Using this pattern, the *observers* does not have to know each other's existence and can be reused on other *subjects*. To use this pattern classes use inheritance to become a *subject* or an *observer*. The *subject observer* pattern used by Sony is slightly modified where an class does not inherit an *subject* or *observer*. Rather, they hold an copy of the *subject/observer* and use it. In addition, unlike the standard version where an *observer* can attach to an subject at run time, in this case the connection are described in the file `connect.cfg`. The figure (11.1)shows a class diagram to clarify the idea. The name Aperios used in the diagrams refers to the Sony OS used for the robot.

The following sequence diagram (fig 11.2)shows how information is passed using the Aperios Subject Observer Pattern. In the diagram the object "Message" is what needs to be sent across.

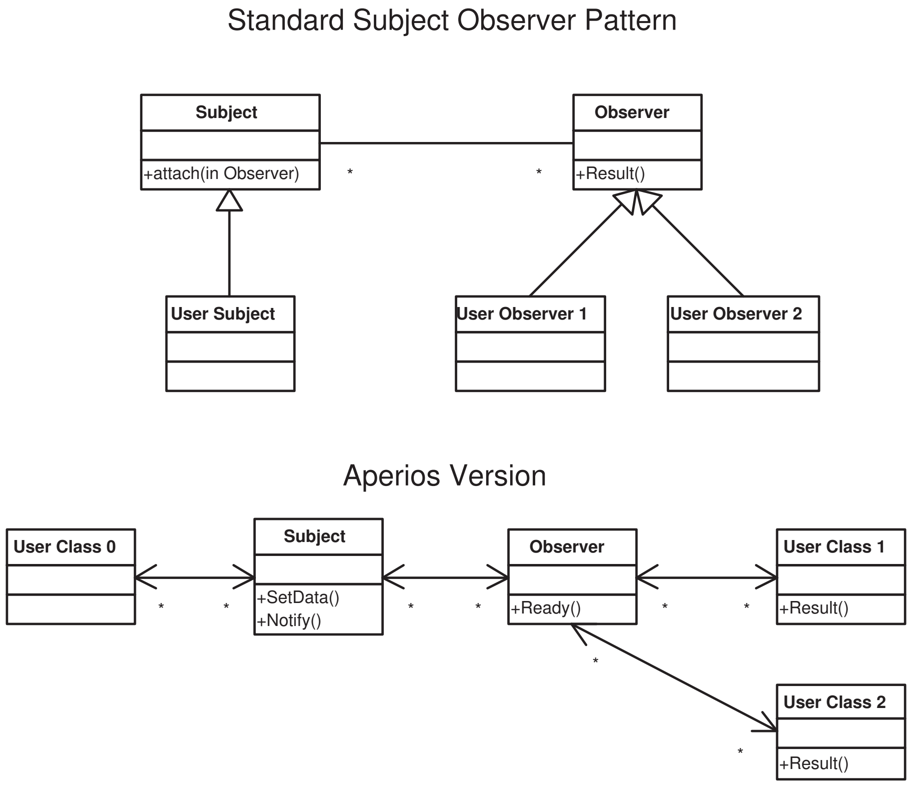


Figure 11.1: Aperios Subject Observer Comparison

11.2.2 Basic Network Infrastructure

Having an understanding of the workings of the Aperios Subject Observer Pattern, we can have a look at the way the wireless communication is conducted. Aperios provides a class which performs network communication via TCP/IP —TCPGateway. We utilise this class to communicate. This is a *subject* and an *observer*. The *subject* part receives information from the wireless network and notifies the *observers* connected to it when data is accepted. The *observer* part waits for information from the *subject* it is connect to, and send any information over the wireless network that it receives. ¿From this understanding, a user have to create an *subject* and *observer* in order to use this service. Each of these objects runs on a separate process. This is similar to the event model in Java.

Sony places a physical constraint in the robot that allows it to only connect to one IP address. This can be another robot or a wireless capable computer. To allow all robots in a team to communicate between each other they must pass the message to a base station (PC running Linux/Cygwin) which will redirect the message on to the destination. *Subjects* and *Observers* are created on both the robots and base station, to facilitate the communication. The fig 11.3 shows

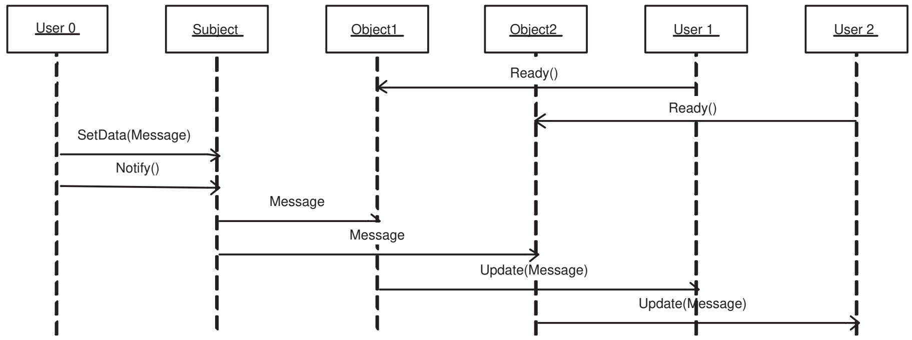


Figure 11.2: Aperios Subject Observer Pattern Sequence

an simplified example of how 2 robots are connected.

In figure 11.3 labels with an extension `.O` such as `MsgPasser.BOT2.CHAR.O` represent an *observer*. While labels with an extension `.S` such as `MsgPasser.BOT2.CHAR.S` represent a *subject*. These labels are also those that you will see in the `connect.cfg` file. This file describes the connections between the Aperios objects and the user (Wireless Class, Message Passer Class). (I have assumed the reader already knows what the connect.cfg files does. Please refer to the past reports for more information.) The arrows in the diagram represents the direction the data traverses.

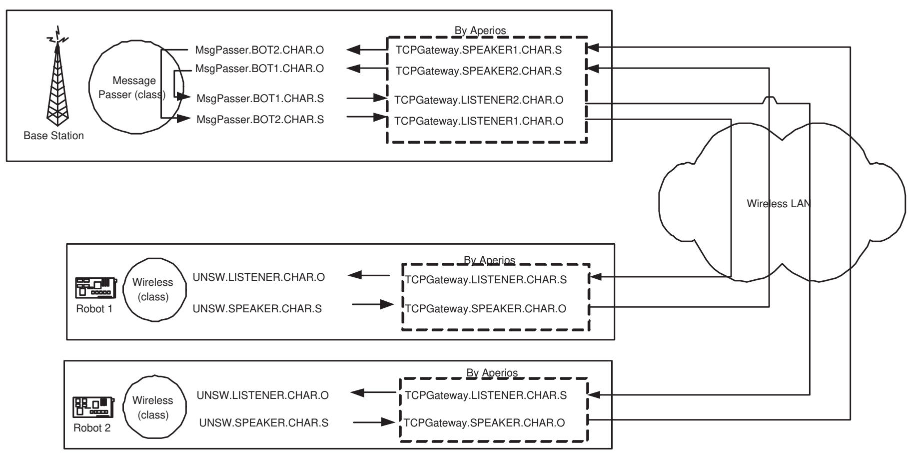


Figure 11.3: Communication Structure for 2 Robots

For a TCP/IP communication to be successfully established, the ports to make a connection also needs to be specified. This information is described in the file `portmap.cfg`. This file contains a number of lines, each line contains 2 pieces of information - the object and the port it uses for receiving or sending. An example can be `TCPGateway.SPEAKER.CHAR.O 1100`. Thus it makes sense to use unique ports for each wireless connection in figure11.3. The details of the ports can be seen in section 11.7.2.

11.3 Message Packet Format

Linking up the robots is a complicated procedure. The complexity does not stop here. The underlying operating system Aperios sends messages in its own message packet format and protocol. In particular, the message is absent of any detail on what content is stored and its corresponding size –the Aperios *subject observer* pattern. For our purpose we would like to send a range of data types such as real time camera frame and the world model. This means, specialised way of communication between the robots is required, whereby for each message dispatched, an extra fixed length message must be sent beforehand, to provide information regarding size and type. This is similar to the idea used in IP —a header for each piece of data. To visualise the problem I refer you to figure 11.4. The header is a fixed size c++ struct named `WDataInfo` which hold information regarding the source robot, type of message and the size of message. The various types of messages and the definition of this struct can be found in `WirelessTypeDef.h`.

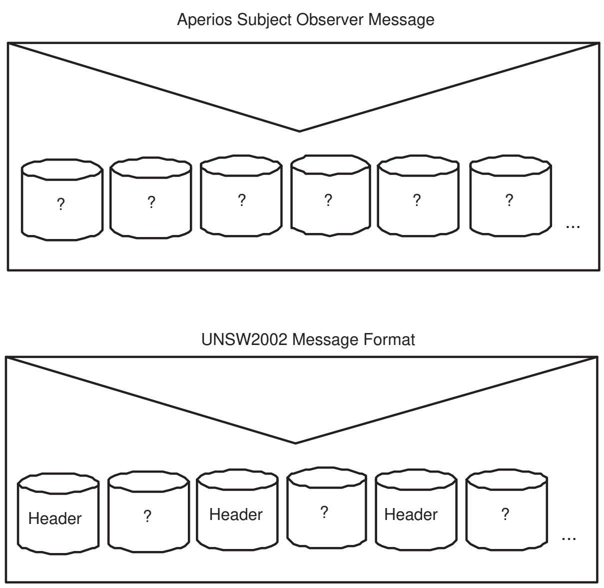


Figure 11.4: UNSW2002 Message Packet Format

11.4 Code Modifications

To allow the robots to use these communication facilities, the structure of the robot programs also needs modification. The method of incorporating this extra functionality is very similar to how the different modules of the program are grouped together —C++ multiple inheritance. By using, multiple inheritance, wireless communication capabilities can be provided seamlessly. Multiple inheritance also allows the variables of the class to accessed conveniently. Figure 11.5 shows the extra class that is added to the existing code structure.

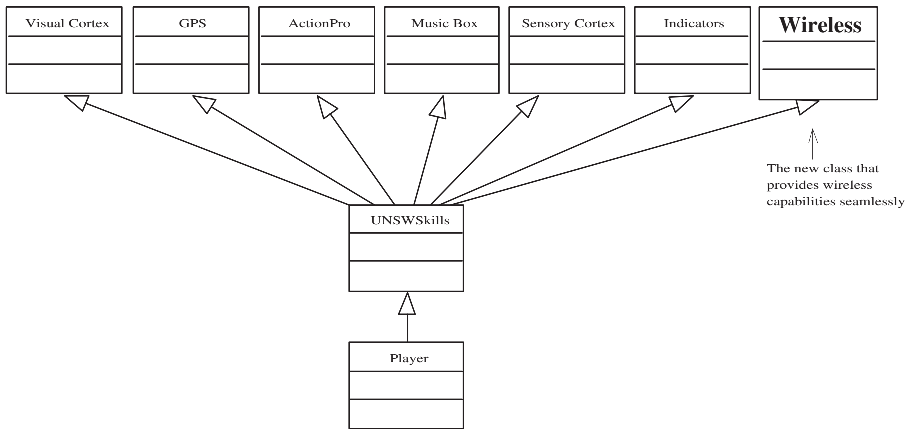


Figure 11.5: Sony Game Controller Connection

11.4.1 Using Wireless Capabilities

Sending information over the wireless is almost as easy as a simple function call. Nevertheless, there are a few small points that needs to be raised. Here I will give a walk through on how to use the wireless.

To send data, you call the method `wsend(void*,int,int)` indicating the data you want to send, the type of the data and the byte side of the data. The type is just any integer value so the programmer can distinguish the message when it is received on the other side. You can call this function multimple times. Once you have finished indicating all the data you want to send, call the method `wflush()` to send the the information over wireless. The information will be received by the base station which may simply forward it to all other robots.

Processing in the robot is started every time a camera frame is ready. During this processing period, you can only call `wflush()` once. This is because if you call `wflush()` multiple times during one processing period, previous data set using `wflush()` will be overridden. To avoid this you can simply call `wflush()` every time at the end of processing regardless wether you have set some data to send. If you haven't set any data, nothing is sent.

When data is received over wireless, the method `UNSWSkills::wreceive` is called for every piece of information sent using `wsend`. This means if a teammate called `wsend` 3 times then your `wreceive` will be called 3 times. This method is called independent to the cameral frame. The sender of the data can be other robots or the base station itself by the debugging tool. You can distinguish the source from a parameter of the method. Presently, code exist that already processes recognised data type such as debugging information and team-mate world model.

11.5 Base Station

The sections above described the wireless component in the robots. Using Aperios, a robot can only connect to one other object. This can be another robot or a computer with wireless capabilities. In a game there are multiple robots meaning a base station is required to relay the messages. A base station is simply a computer that uses the Aperios libraries enabling them to receives messages from robots and process the messages, which can be a simple forward function. Using a base station also provides developers with a set of excellent debugging tools. In here I will outline the base station I have developed. The debugging tools will be described in chapter 12.

The base station uses the Aperios library. This means it adheres to the *subject observer* pattern. There is a `connect.cfg` file that describes the connection between the TCPGateway and the program. The concept of the connections is visualised in figure 11.6. For each robot there are 2 TCPGateway objects created. One for sending one for receiving. We also create an *observer* and a *subject* object for each robot in the code. The *observer* connects to the corresponding TCPGateway *subject* that receives data from the robot. The *subject* on the other hand connects to all the other TCPGateway *observers* that sends information to robots excluding the TCPGateway sending information back to the same robot.

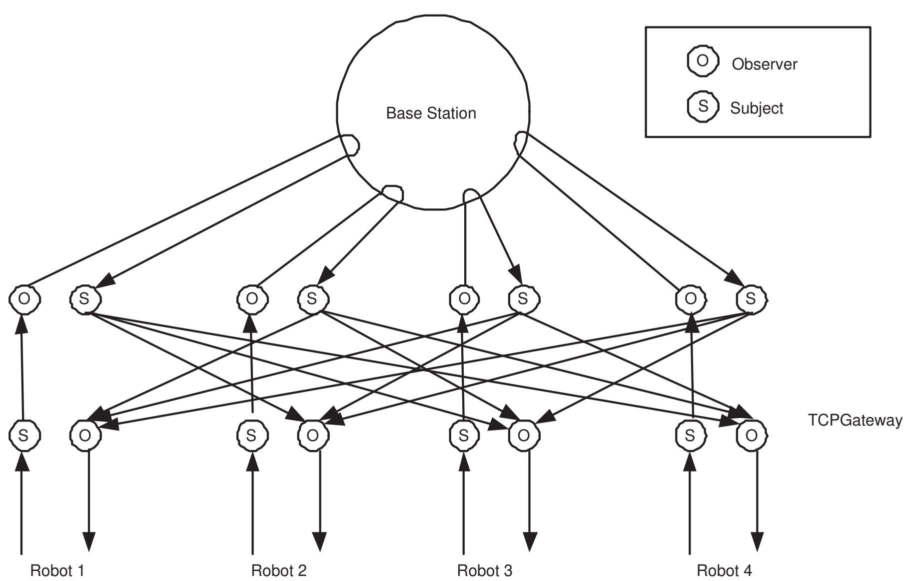


Figure 11.6: Base Station Connection Model

¿From the conceptual diagram in figure 11.6 we can see that the base station simply forwards the messages. The c++ class in the base station that does this is called `MsgPasser`. The base station is actually integrated with the debugging tool "RoboCommander". Please refer to the chapter on debugging tools for a full description.

11.6 Game Controller

In the previous years, the robots are started and stopped via tapping on their heads. This is places strain on the fragile head motors. The introduction of wireless into this year's RoboCup competition means that robots can be started and stopped through a remote game controller. Sony has written a wireless game controller to be used in the competition. To cater for this system, modifications were made to the existing communication structure. An extra connection is made from the robot to the base station which receives the game control information. In this process an extra TCPGateway object was created and an extra *observer* was allocated to receive the game commands. This was put in place, on top of what we already have.

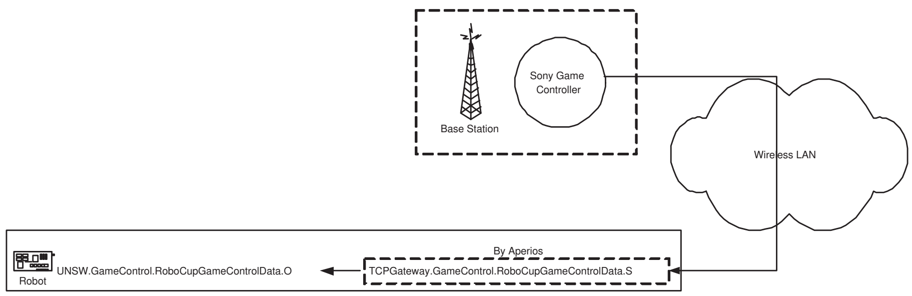


Figure 11.7: Sony Game Controller Connection

11.6.1 Game Controller Robot Communication Structure

Sony has created their own network connection model to be used in the games that is different to our connection model. This requirement was endorsed by Sony to ensure consistent connection model, as well as for purpose of traffic monitoring, amongst other reasons. During game, communication between the robots are conducted by communicating to the Sony game controller. This controller will then route the messages between the robots. The teams are free to send any form of information. The controller simply copies the message to all robots.

In our communication model, each robot only have 2 connections to the base station. One for send and one for receive. A robot identifies the sender of the message from the header. The base station merely broadcasts the message by giving a copy to all team-mates. Hence in total there are only 8 connections for a team of 4 robots. On the other hand, the way Sony's network structure works, requires that each robot have a sending connection to the base station, at the same time having 3 receiving connections. One for each team-mate. This way the source of the message can be implicitly found. The conceptual difference is show in figure 11.8.

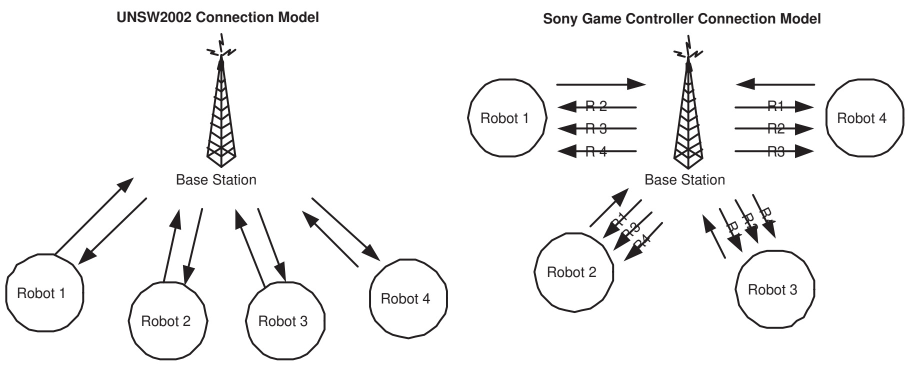


Figure 11.8: Sony Game Controller Connection

During a match only world model information is sent over the connection to Sony game controller. All other information is sent via our own connections. This separation also prevents the accidental broadcasting of debugging messages to team-mates.

11.7 Aperios Subject Observer Details

11.7.1 Subject Observer Connections

The follow tables the connections between TCPGatway and the *subject* and *observers* used by our code. This contains the our network connections and those defined by Sony.

Subject	**Observer**
UNSW2002 Network Connections	
HAR.S UNSW.DogComm.CHAR.O	TCPGateway.SPEAKER.CHAR.O
TCPGateway.LISTENER.CHAR.S	UNSW.LISTENER.CHAR.O
Sony Game Controller Connections	
TCPGateway.GameControl.↩	UNSW.GameControl.↩
→ RoboCupGameControlData.S	→ RoboCupGameControlData.O
UNSW.DogComm.CHAR.S	TCPGateway.OutgoingComms.CHAR.O
TCPGateway.Robot2.CHAR.S	UNSW.DogComm.CHAR.O
TCPGateway.Robot3.CHAR.S	UNSW.DogComm.CHAR.O
TCPGateway.Robot4.CHAR.S	UNSW.DogComm.CHAR.O

11.7.2 Port Connections

The following is a table of the ports each TCPGateway uses. This is defined in the file `portmap.cfg`

TCPGateway	**port**
UNSW2002 Network Connections	
TCPGateway.SPEAKER.CHAR.O	1100
TCPGateway.LISTENER.CHAR.S	1200
Sony Game Controller Connections	
TCPGateway.GameControl.RoboCupGameControlData.S	1070
TCPGateway.OutgoingComms.CHAR.O	1101
TCPGateway.Robot2.CHAR.S	1202
TCPGateway.Robot3.CHAR.S	1203
TCPGateway.Robot4.CHAR.S	1204

11.8 Experiments and Results

The wireless was an excellent tool especially for debugging. However, during initial tests we found some debugging messages were lost. Even when there is only one robot on the field, some messages from the robot was lost. This has lead to the investigation to the reliability of the wireless capability.

11.8.1 Reliability, Latency and Bandwidth

To test the different properties of the wireless capability such as speed, reliability and bandwidth, a simple player (Ping) was created and installed on 2 robots. One robot simply sends a small dummy message at every camera frame to the other robot and the receiver simply bounces the message back. Latency can be found by comparing the time stamp of the returned message. There are a number existing tools that analysis the traffic over a wireless network, however, the aim of our test is the ability of the robots to utilise the wireless and its behaviour. Hence, the wireless traffic or latency is best to be observed in the robots for a realistic feed back in gaming conditions. Large number of samples were collected for this experiment. The summary of the results found is described in the following table.

Distance	close (<10 cm)	on field (<5 m)	far (<10 m)
Average Latency (to and back)	438.1333 ms	440.5789 ms	464.7643 ms
Average Drop Rate	4.114094	4.986755	3.980769
Peak Drop Rate	6	6	6
Lowest Drop Rate	1	4	2

In the above table the distance describes the length between the robot and the base station. The average latency describes the latency for the packet to reach to the other robot and come back to the sender. The average drop rate is the average number of messages missed between consecutive returned message.

For example, a value of 4 means 3 message are dropped and only the fourth message is returned. In here I describe the dropping of information in units of messages in stead of the usual TCP/IP terms of packets, because when we send information over the wireless network we actually send messages to the Aperios object TCPGateway. TCPGateway holds its own message queue. This is a queue of only one length. If a previous message has not completed in its sending, the second call to the object will place the message on the queue, a third call will erase the previous message in the queue and replace it with the current message. This result causes message dropping. The observations in the table tell us that the latency increases as the robot gets further from the base station. However, the increase in latency is not significant. This means latency is quite consistent regardless of distance. The drop rate is a little peculiar, when the robot gets far the drop rate seems to be smaller on average. After a number of further tests I found the figure are affected by the environment of the robot. If there is an object right between the robot and the base station there is higher drop rate. Since the base station is located behind the fence of the field, this may cause the robot on the field to have a higher drop rate than if it was about 10 meters away in a line of sight. Under these considerations the average drop rate is not a representative figure of the reliability. Instead we should use the peak drop rate as a guidance to a safety margin we should use as the rate of sending messages.

Using the same tool I have also tested sending the largest message during game play. This is the full world model information. Testing is achieved by loading a standard player on four robots and program each robot to send their world model to team mates at every 6 frames. Under this test I have found there was very little and almost no message drops. For testing purposes I have also attempted to send CPlane information of every robot to the wireless base station. Unfortunately when all 4 robots were sending, it flooded the wireless network and very few frames were seen on the base station. Under another test, we only allowed one robot to send the CPlane. We found the best interval for sending CPlane is 7.

11.8.2 Effects of Wireless on Robot Behaviour

Sending messages over the wireless network uses the TCP/IP protocol stack in the TCPGateway object. If the messages are too large then this object gets too much processing time allocated. This reduces the processing time on the actual strategy object. In result this will slow the robot. To maximum the time allocated to the processing of strategy object, tests were conducted during game play that records the average number of camera frame processed per second and the number messages dropped. The robot's camera is able to capture 25 frames per second. The table following summaries all the findings.

	Sending Interval	Dummy Message	World Model	CPlane
Size (Bytes)		12 bytes	640 bytes	6336 bytes approx
Message Drop (All)	6	0	0	3 - 4 approx
Message Drop (Single)	6	0	0	1 - 2 approx
Message Drop (Single)	8	0	0	0
Camera Frame Processed	6	25	24 - 25 approx	20 - 23 approx
Camera Frame Processed	8	25	24.5 - 25 approx	20 - 23 approx
Camera Frame Processed	12	25	25 approx	22 - 24 approx

In the above table, 'Message Drop' gives the number messages dropped when sending messages using the frame interval described in the Sending Interval column. This is tested in the case when all the robots are sending (All), and it is tested when only one robot (Single) is sending. Many of the values such as the the number of processed camera frame is approximated because, they are captured during game play which depends on the surrounding environment of the robot. If the robot sees lots of objects (blobs) then the processing will be slowed down by vision and in effected it will reduce the number of frames processed. The size of the packet for the CPlane is also approximated because we applied Run Length Encoding to compress the CPlane and the effectiveness of the compression can vary depending on the image. From the data in the above table, we can see that, if we send world model information at every 6 frames, then there is almost no message loss. However, this will reduce the number of camera frames processed slightly. For the competition, we have prepared our strategy to only send world model information at every 12 frames. This was chosen to give maximum processing time to the strategy object and minimise any unforeseeable processing overheads that may occur.

11.9 Conclusion

The wireless capabilities have give many opportunities to explore different possibilities in the RoboCup2002 competition. It allowed robots to share their world model. Wireless also made debugging much easier. We have used our unique connection model to implement the wireless capability and successfully solved many problems such as message identity. We have also integrated the communication to the Sony Game Controller allowing the robots to be controlled and communicate under the Sony connection model.

The wireless has limitations in terms of latency, reliability and it may affect the processing of the robot. Under a number tests, figures where found that will guide the sending interval and the amount of data that can be sent. They will help developer to correctly utilise this capability.

Chapter 12

Wireless Debugging Tools

Section author: Zheng David Wang

12.1 Introduction

The introduction of wireless into RoboCup has opened many possibilities. It's convenience of setup and usage, it's fast transmission speed compared to the old serial cable, and it's flexibility to send any data across the wireless network has motivated the development of a number of wireless debugging tools. This includes the tools to print out debugging statements, tools to display what the robot sees and tools to display the world model. Having these tools can make development much faster and easier to debug. They can give the developer real time feed back of the robots, visualisation of the state of the robot and real time manual control of the robot.

12.2 RoboCommander / Base Station

To realise all our great ambitions for the wireless technology the simple base station is upgraded to act also as an debugging tool, hence the name RoboCommander (figure 12.1). It provide simple and yet useful functions. The center of the dialog displays the debugging messages printed from the robots. The two text field at the bottom of the dialog allows the sending of a name and value pair to the robot. This allows the robot's behaviour to be modified in real time. Finally, the "Motion" button at the bottom right of the screen allows the robot to be controlled with the keyboard. These functions will be described further later.

To communicate to the robots, we must use the Aperios libraries. This had to be written in a c++ module. RoboCommander (GUI) on other hand is written in Java. Java was used because it was faster and easier to develop graphical

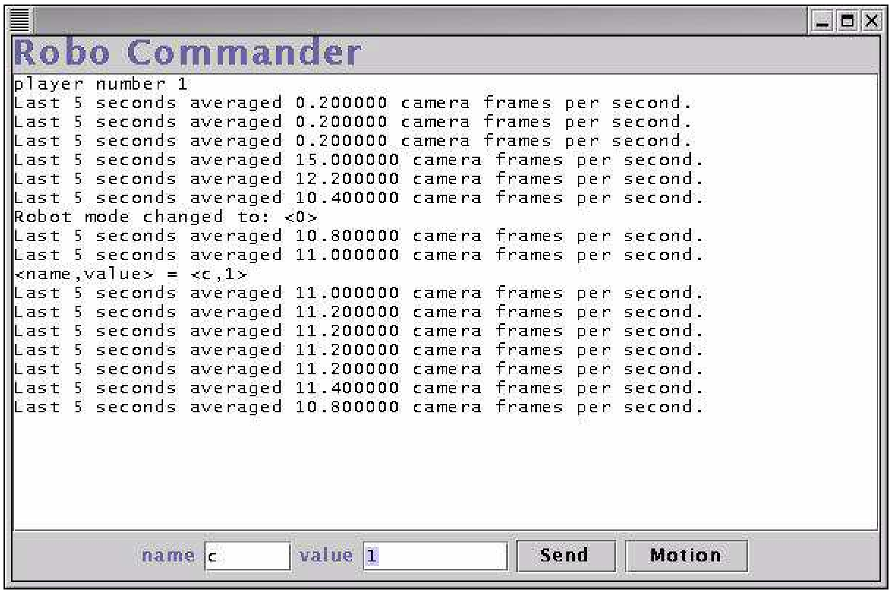


Figure 12.1: RoboCommander

user interfaces with the Java Swing library. In order for RoboCommander to communicate to the c++ component, an internal TCP/IP connection is made between the two. In addition, to allow the c++ component to receive messages from the Java interface, a separate *subject* —`MsgReceiver`, had to be created which constantly listens for messages from the Java side. This object is connected through Aperios to an *observer* in `MsgPasser`. `MsgPasser` is the c++ class which sends, receives and relays messages between the robots (see section 11.5).

12.3 `cout` and `wout`

Often in debugging, one of the most useful tools is `cout`. However, using `cout` on the robots from Sony is expensive, as information is transmitted via the slow serial cord on to a serial debugger. Consequently, this would have slown down your program if you have many print statementsand subsequently slow down the robot. In addition, the serial cable attached to the robot made debugging during game play almost impossible. With the powerful wireless communications developed, one of the first upgrade is the development of a `wout`.

`wout` is an object in the Wireless class that can be used just like `cout`. It transmits the debugging messages over the fast wireless network and displays them on the base station. The debugging messages will be printed in the large text area in the middle of the dialog (see figure 12.1). One thing to remember, when you use `wout` is that data are not immediately sent over the wireless.

They are stored until `wflush()` is called. For convenience, you can simply call `wflush()` at the end of all processes. This is a safe method. The Wireless class will simply ignore the call when there is nothing in the buffer to send. The following is a code example that shows how easy it is to use `wout`.

```
cout << "robot x " << rx << " robot y " << ry << "\n";
```

12.4 Wireless Control

`wout` is a excellent debugging tool. Being able to send information to the dog is equally important. This will allow modification of program behaviour in real-time without changing the code and rebooting the dog, which is a time consuming process. Imagining recompiling the code and then load the memory stick in the robot every time you try a new parameter. This can take a long time. With the wireless tool your can modify those values easily. You simply enter a name and value pair in the text fields of RoboCommander and click "Send". This will also allow the robots to be remotely controlled. One of the useful functions of this tools is you can set the mode of the robot defined in `UNSWSkills.h` by typing "mode" as the value and an integer as the mode you want to set. This allows the robot to be remotely started or stopped. The following is a table of useful modes.

Name	Value	Description
mode	21	stationary localise
mode	22	demobilized, stops the robot
mode	0	normal, starts the robot

Table 12.1: Common Modes

12.4.1 WirelessControlExercise.cc

This is a teaching player (main strategy code) file that shows you how to use the wireless capabilities in the robot. The existing code allows you to control the robot's movement, display its *CPlane* and display its *World Model*. This file contain a method named `processWControl` which demonstrates how to make use of the information sent over via RoboCommander.

With this file you can make the robot walk forward, left, strafe, turn, move it's head, etc. Each movement is defined by a name and value pair which can be entered in the text field of RoboCommander. To make the issuing of these common commands easier, there is a "**Motion**" button on RoboCommander that allows you to control the using keyboard strokes that are commonly used in 3D shooters (Counter Strike or Quake). Once the button is the clicked the keyboard strokes listed in the following table will move the robot. I have implemented sticky keys. That means to stop the robot you must reverse the motion by pressing keys that are exactly opposite to the current motion.

Keyboard	Function	Name	Value	Value With Shift Key	Unit
w	move forward	f	4	2	cm/s
s	move backward	f	-4	-2	cm/s
a	strafe left	l	4	2	cm/s
d	strafe right	l	-4	-2	cm/s
q	turn left	t	20	40	deg/s
e	turn right	t	-20	-40	deg/s
i	tilt head up	v	5	-	deg/s
k	tilt head down	v	-5	-	deg/s
j	pan head left	h	5	-	deg/s
l	pan head right	h	-1	-	deg/s

Table 12.2: Keyboard Strokes to Control Robot

12.5 World Model Display

One of the most difficult components to debug in the program is the world model. Unlike vision where results of object recognition can be directly checked by looking at the still image from the camera, World Model can not be checked in such a simple manner. It is of course possible to simply print out the co-ordinates of where the robot thinks it and the rest of the object is, but because updates are performed at 25 times a second, the print statements are too fast to be readable. Also, localisation will always be an approximation; a more appropriate debugging tool would be to display the world model of the robot in a graphical form where we can easily verify the location of the robot on the field. This tool also allows us to see the jitters in the localisation, which could be a result of noisy visual recognition.

The world model display does not only show the world model but also the model from vision and the wireless model. The different types of models can be activated by clicking on the buttons located at the right hand side of the display. Different model can be displayed simultaneously. The vision model uses lighter colours for ball and robots, while the wireless model uses darker colours for ball and robots. In the world model display, red and blue solid circles indicate red and blue robots correspondingly. The robot itself is displayed with 2 black outline circles. The arrow on the circles correspond to the robot's heading. At the moment, the heading recognition of robots from vision is quite unreliable, hence only the heading of the robot itself, and the heading of the team-mate robots in the wireless model is updated. In the world model display, each robot has a surrounding circle, that circle indicates the 95% confidence region of the robot. In other words distance from the circumference of the circle to the center of the robot equals $2 * \sqrt{variance}$. The orange circle in the display represents the ball. It also has a surrounding circle indicating its 95% confidence region.

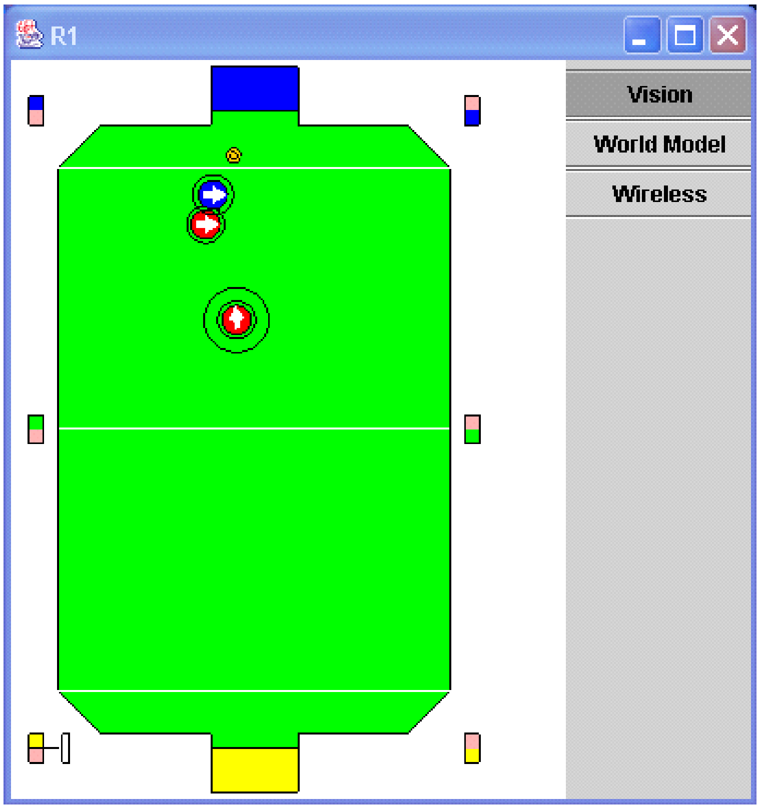


Figure 12.2: World Model Display

12.6 Colour Plane Display

In order for the robots to recognise objects in its surroundings, each colour pixels of a 24-bit image from its camera is classified into a small number of specified colours. This allows objects to be easily identified. Distance measurement is heavily dependent on the number of correctly identified pixels. A simple miss classification can easily affect the distance measurement. To combat that, it is preferable to display what the robot sees in real time. This would not have been possible with the old serial debugging tools which took about 3 seconds to display one image. Using wireless it allowed displaying of a series of images rapidly. It gave opportunities to catch glitches that would be missed via observation of still images from the robots. See figure 12.3.

The colour plane displays the classified image and the bounding boxing around each identified objects. At the corner of the bounding box there is a number indicating the object number that has been identified. These number can be referred to the c++ class `VisualCortex` to determine the object identified. Each bounding box also has a horizontal and vertical line across the center extending to the edges. This cross is placed there on purpose to determine the size of bounding boxes that might have parts outside the display. In this display there

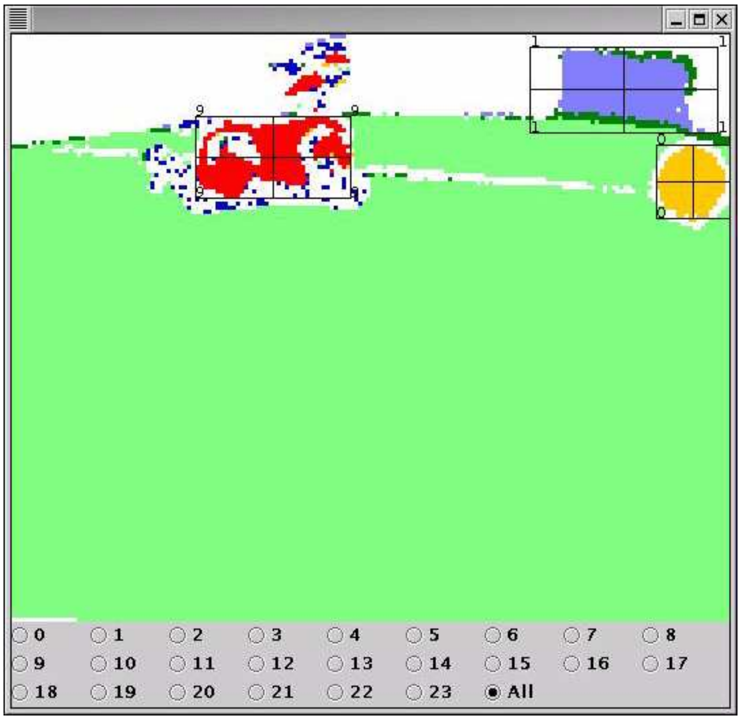


Figure 12.3: Colour Plane Display

is a series of radio buttons at the bottom. It allows the user to selectively display only the selected colours.

12.7 Remote Painting Program

Classifying a 24-bit pixel to a fixed category requires colour learning process which can take time. Sometimes when you observe that small amounts of pixels are misclassified and wish to fix it, the whole learning process must start over again. This can prove to be both time consuming and frustrating when you find the new classification to be worse off. To dramatically improve this process, a painter program has been created that allow colour classification to be modified in real-time. This allows corrections due to minor misclassification to be carried out quickly. It also allows the capture of photos from the robot's camera. See figure 12.4.

When you click on a pixel on the display the YUV value of the pixel is sent to the robot to classify that colour as the colour category indicated by the radio button at the bottom of the dialog. Once a pixel is clicked it also colours all pixels on the display, with the same YUV value using the specified colour indicated by the radio buttons. The "View Scale" field allows you to scale the size of the

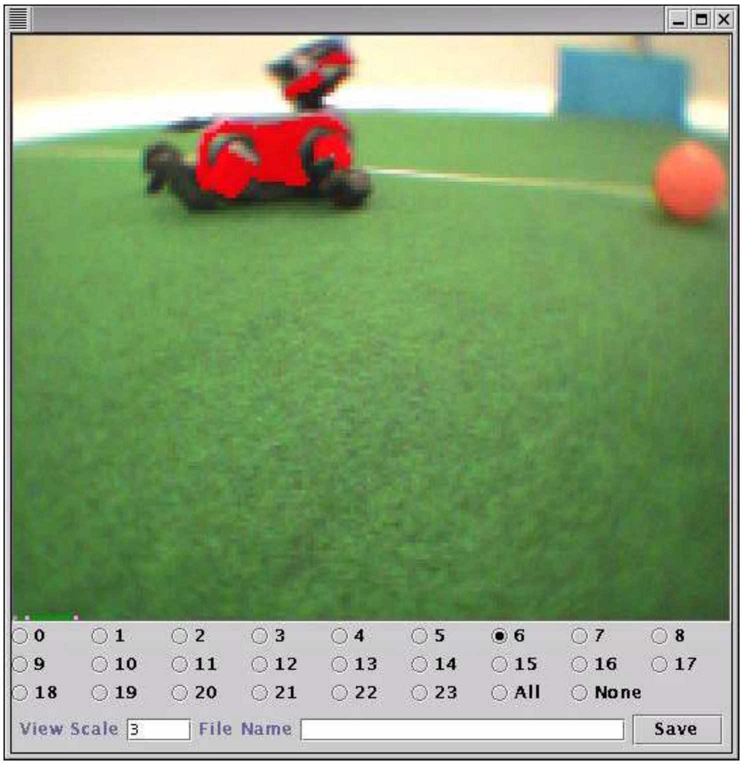


Figure 12.4: Colour Plane Display

image. The “File Name” field allows you to save the displayed photo in a file. The format of the file saved is in BFL format. Each row of pixels generates 4 lines of integers separated by spaces in this file, a line for each of Y, U, V and colour class. For more details of the format of the file please refer to the chapter on vision.

Chapter 13

Results

Chapter author: Min Sub Kim.

13.1 The Australian Open

The Australian Open was not an official part of the RoboCup competition, but was a friendly competition organised by the three Australian universities competing in the legged league (UNSW, Newcastle University, and Melbourne University). It was held on the 27th of May, just three weeks prior to the actual RoboCup competition, at Newcastle University.

Each team played the two other teams twice during the day, for a total of three games.

In the first game, we played Newcastle University and won 13 - 0. Newcastle seemed to have trouble with their goalkeeper for most of the first half.

In the second game, we played Melbourne University and won 6 - 0. We noticed that our scoring rate was slowed considerably by their swarming strategy, causing some concern.

In the last game, Newcastle played Melbourne, and drew 3 - 3. The match was decided on a penalty shootout, won by Melbourne.

The Australian Open was a good practice competition for our team to see where in stood for the competition, as we were finally able to play against other teams.

13.2 The RoboCup Soccer Tournament

The soccer tournament is the main competition in the legged league, and spans most of the competition week.

The competition was held this year from 17th June to 25th June, with the symposium occupying the last two days. The venue of the competition events was the Fukuoka Dome Stadium.

One of the differences between the competition this year and previous years is that the venue was closed at night, except over the first setup day. This does reduce the time available to make any last minute fixes, but must be realised that it is realistically impossible to make any dramatic improvements by working around the clock during the competition week. Another difference is that the symposium was held at the end of the competition this year, whereas in previous years it has been held in the middle of the competition week. This also reduced the time available for late development.

13.2.1 Team Setup, Monday 17th June

Having arrived in Japan the previous day, we spent most of the setup day setting up our work environment and just making sure that everything was still working as expected. We noticed that the competition field was different to the field we have at our lab, in that the lighting conditions and carpet were slightly different, but generally everything seems to be in order.

The teams were divided into four groups on the day. With 19 teams, this led to 3 groups with 5 teams and 1 group with 4 teams. The top four teams from last year were placed into separate groups, with the three new teams for this year also placed into separate groups. Having won the competition last year, UNSW was placed in the 4 team group, with Kyushu, Washington, and the Balkan teams.

The remainder of the day was spent on colour calibration.

13.2.2 Practice Matches, Tuesday 18th June

Prior to commencement of round robin matches, practice matches were held. The practice matches were shortened to just one ten minute half, but it was nevertheless a good opportunity for catching any remaining issues.

We played three practice matches on the day, two against Newcastle and one against Team Sweden. We won each match 5 - 0, with no significant problems prevalent.

We also spent some of the day observing other teams, to get a feel for the competition. Another meeting was also held to conclude the day, in which Sony clarified the last remaining rule questions that had came up in the practice matches.

13.2.3 Round Robin 1, Wednesday 19th June

We played our first round robin match against Kyushu. We began strong in the first half, racing out to 5 - 0 by the half time break, but had a slow second half, finishing 7 - 0. We noticed that our robots seemed lost near the side edge of the field, and suspected that it may have been caused by incorrectly identifying objects outside the field, since there was some glare on the barriers around the field.

On later investigation, we were not able to reproduce or identify the error from the match. The error did not present itself in the remainder of the tournament either.

13.2.4 Round Robin 2, Thursday 20th June

We played our second round robin match against the Balkan team, and won 16 - 0. The winning margin was the largest ever in the legged league.

We did not notice any significant problems with our robots in the match.

13.2.5 Round Robin 3, Friday 21st June

Our last round robin match was played against Washington, and we won 10 - 0.

Having completed the round robin matches without conceding a single goal, and with a higher goal total than any other team despite being in a smaller pool, we headed into the quarter finals with confidence.

13.2.6 Quarter Finals, Saturday 22nd June

The quarter finals were held on Saturday, interleaved with the challenge competition.

We played our quarter final match against the German Team, and had a very shaky start, leading 1 - 0 at the break. Our forwards were penalised several times for charging their goalkeeper, as they had adopted a strategy of deliberately holding onto the ball with the goalkeeper. This slowed our scoring rate considerably. Our goalkeeper also had issues in the first half, often wondering far out of position. Fortunately, our opposition was unable to take advantage of the problem.

In the second half, we played much more strongly and finally won 6 - 1. After the match, we tried to find the cause of the poor play in the first half. We suspected that somehow the shared network was slowing our robots down, and decided to play the remaining games with wireless completely disabled.

In the other quarter final matches, CMU defeated Team Sweden 9 - 0, Melbourne defeated Kyushu 3 - 0, and Newcastle defeated USTC 3 - 0. It was encouraging to see that all three Australian teams had made the semi finals.

13.2.7 Semi Finals, Sunday 23rd June

We played our semi final match against Newcastle. It was a good match throughout, and we came out on top 3 - 0. Newcastle had done a lot of work in the weeks since the Australian Open to cut the difference by that degree.

In the other semi final, CMU defeated Melbourne 4 - 0. Thus, we were heading into the finals against CMU, while Newcastle and Melbourne would play for 3rd place.

13.2.8 Finals, Sunday 23rd June

We began slowly in the finals, and CMU jumped out to an early lead, scoring the first two goals. We managed to score one in return before the break, heading into the second half down 2 - 1.

In the second half, we started out strong scoring two goals, to move ahead 3 - 2 with a few minutes remaining. CMU scored one in return, making the score 3 - 3 with a couple of minutes remaining. No more goals were scored and the match went into a penalty shootout.

In the penalty shootout, we were narrowly defeated 2 - 1. The penalty shootout takes a sudden death format, and we missed the second shot, failing to score in the given minute. In general, our penalty shooter seemed to suffer from vision and localisation problems.

In hindsight, it would have been preferred that the penalty shooter had occupied more development time, but we were not able to test the penalty shooter extensively with the tight development schedule.

The 3rd place match also went down to a penalty shootout, after Newcastle and Melbourne tied 2 - 2. Newcastle won that shootout 2 - 1.

13.3 The RoboCup Challenge Competition

The challenge competition is a separate competition held in the legged league, in which teams demonstrate skills that may not necessarily be directly related to the soccer competition. This encourages research into other fields that are not covered by the soccer competition.

We won the challenge competition overall, winning two out of the three challenges, and placing second in the other. The results and details of each individual challenge are covered in the challenges chapter.

The overall results are summarised in the table below. Teams are awarded points for each challenge based on their ranking, thus the point difference in the challenge results is sometimes not reflective of the actual difference in performance.

Ranking	**Team Name**	**Challenge 1**	**Challenge 2**	**Challenge 3**	**Scores**
1	rUNSWift	18	19	19	56
2	CM-Pack'02	19	18	18	55
3	NUbots	16	16	17	49
4	RoboMutts++	15	15	16	46
5	Upennalizers	12	12	14	38
6	German Team	13	8	15	36
7	Wright Eagle2002	11	17	8	36
8	UW Huskies	17	9	9	35
9	ARAIBO	7	14	11	32
10	McGill Red Dogs	7	10	10	27
11	Baby Tigers	14	5	6	25
12	SPQR	0	11	13	24
13	Team Sweden	7	13	4	24
14	ASURA	7	3	12	22
15	Essex Rovers	7	7	4	18
16	Les 3 Mousquetaires	7	6	4	17
17	Cerberus	7	4	4	15
18	Borregos Salvajes	0	0	7	7
-	Yellow Jackets	-	-	-	-

Chapter 14

Conclusion and Future Work

The Robocup project presents and will continue to present many areas for further improvement and development as the Legged-League Competition progresses. The competition is persistant in leading areas of artificial intelligence into practical application and testing. The areas for improvement and further work that extend from this report have been discussed in detail within each chapter. Following is a summary of these areas for improvement and possibly the starting point for work in the next year and beyond.

Firstly though, there were some issues which were raised when the team was competing in Japan. A change that was implemented very close to the competition time was believed to have contributed to some of the unwanted results. The change was that for the ball to be recognised as a ball by the dog, the dog needed to have classified it as a ball for two consecutive frames. This was implemented to solve the problem of the reflection of the Red dog on the Yellow goal which often caused the dog to recognise a ball in the goal. However this change had an effect on the exit condition of the *findball* routine as when looking quickly for the ball the dog would often only see it for 1 frame, and so the dog would not exit the routine as it had previously. This is likely to have caused a poorer in-game performance of the system.

The Goalie Player Strategy has a large area of work where further improvement can take place. This improvement is particularly in the area of the Less Aggressive Goalie. Next year there is the possibility of the introduction of a change in rule concerning the penalty area around the goal. This area may be changed to be a rectangle shape around the goal, removing the need for the goalie to be able to navigate into the corner of the field and back to the centre of the goal. With this change in rules there is even more room for advancement and for the assembly of a complete, successful and reliable goalie strategy.

Locomotion was an area that was explored quite extensively this year. Many new kick's were experimented with and a new walk style was introduced which gave the dogs a helpful speed advantage. Despite the work that has gone into this area of research, there is still room for more development. The future research is focussed in three areas: walk styles, kick styles and the get-up routine. Research into the area of the walk style may involve having walk styles that cooperate with other areas of the system. This would be a great advantage, for instance, walk styles with less body movement (allowing better vision), with faster dog movement or for careful ball approaching could improve gameplay significantly. Different methods of propelling the ball could be explored further. Varying the speed of the limb movement during the kick process is just one way that better distance and accuracy yields may be reached. The untouched Sony standard *get-up routine* can always be improved to be faster and more reliable.

A frustration for those working wit the Vision system this year was colour calibration. The taking of the calibration images using the dog and then the manual colouring of those images once they were taken took a substantial amount of time. Automating all or even only part of the process would be an area in the future that almost certainly needs to be explored. Aside from this, a totally different approach to the vision system may be rewarding. The current vision system is heavily affected by lighting conditions. Perhaps a system based more on edge detection and contrast would be less drastically affected and even allow better object recognition.

The recognition of objects in a scene is highly affected by the quality of the scene that is being examined, hence object recognition is largely affected by the Vision system. Improvements in the Vision system will most likely also bring improvements to the Object Recognition. However, a different approach used this year in determining distances to objects was to find out where an object was in the scene then use information about the position of the camera to infer the distance to the object. This was highly affected by dog movement. It may be possible to correct some of the errors in calculation with the inclusion of information from the Locomotion model such as the position of the legs in the Pwalk cycle.

Despite the many improvements obtained through the use of a principled technique such as the Kalman Filter. Localisation remains a difficult problem. Erratic positioning and frequent errors can be reduced somewhat, but not eliminated. Combining information from other robots can be of use on some occasions, but in others situations, it is better to ignore the information coming from team mates and rely on onboard sensors. Thus, many challenges remain in finding effective methods for dealing with uncertainty and noise in physical systems. There are many ways we can improve the localation. We can extend our Kalman Filter to be multidimensional. We can also add methods to estimate the velocity of the object. Next, we can further tune the ad hoc variance constants or perhaps automatically tune them. Further more, we can improve the goal localisation using the beacon-goal method. Finally, we can use the edges of the objects detected for localisation.

Lastly, the Simulator. The largest improvement to the Simulator would probably take the form of a full physics engine to model the interactions of the dogs, the ball and the walls. This would remove some problems that exist with it currently and would also make various additions, such as angled walls, much easier. More complete modeling of the dog body itself would allow a more realistic simulation of the interactions that take place on the field. A greater range of sensors such as vision information and wireless communication along with the ability to add noise to these inputs is another useful addition that could be made. These sensors and their controls, along with the complete modelling of the dog would give a more accurate and thorough test-bed for strategies that are tested within the Simulator environment. With any project, especially in simulation, there is always the question of how far to go and working out where more work with provide the increased gains can be difficult exercise.

List of Figures

Bibliography

[EG95] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[Eks84] M.P. Ekstrom. *Digital Image Processing Techniques*. Academic Press Inc., 1984.

[Fai88] M.C. Fairhirst. *Computer vision for Robotic Systems - An Introduction*. Prentice Hall, 1988.

[HFT+99] G.S. Hornby, M. Fujita, S. Takamura, T. Yamamoto, and O. Hanagata. Autonomous evolution of gaits with the sony quadruped robot. *Genetic and Evolutionary Computation Conference*, 1999.

[HG98] M. Huber and R.A. Grupen. A control structure for learning locomotion gaits. *7th International Symposium on Robotics and Applications*, 1998.

[HIPS00] B. Hengst, D. Ibbotson, S.B. Pham, and C. Sammut. The unsw united 2000 sony legged robot software system. *Computer science honours thesis, University of New South Wales*, 2000.

[HIPS01] B. Hengst, D. Ibbotson, S.B. Pham, and C. Sammut. Omnidirectional locomotion for quadruped robots. *RoboCup Internal Symposium*, 2001.

[HTY+00] G.S. Hornby, S. Takamura, J. Yokono, O. Hanagata, T. Yamamoto, and M. Fujita. Evolving robust gaits with aibo. *IEEE International Conference on Robotics and Automation*, 2000.

[JC65] J.W. Tukey J.M. Cooley. An algorithm for the machine calculation of complex fourier series. *Mathemetics of Computation*, pages 297–301, 1965.

[Lew02] M.A. Lewis. Gait adaptation in a quadruped robot. *Autonomous Robots*, 2002.

[MCY02] Fredrik Heintiz ZhanXiang Huang Sprios Kapetanakis Kostas Kostiadis Johan Kummeneje Itsuki Noda Oliver Obst Pat Riley Timo Steffens Yi Wang Mao Chen, Ehsan Foroughi and Xiang Yin. *Robocup Soccer Server, Users Manual*, August 2002. http://sourceforge.net/docman/display_doc.php?docid=11896&group_id=24184.

[Ola02] A. Olave. Research on playing soccer using quadruped robots. *Computer Engineering thesis, University of New South Wales*, 2002.

[RCG] Richard E. Woods Rafael C. Gonzalez. *Digital Image Processing*. Addison-Wesly Publising Company, first edition.

[Rus94] J.C. Russ. *Image Processing Handbook*. CRC Press Inc., second edition, 1994.

[spi87] Review of machine vision and architectures. *SPIE Proceedings*, January 1987.

[TB00] Brian Chemel Rosemary Emery David Huggins Maria Hybinette Scott Lenser Juan Carlos Manuela Veloso Ron Arkin Will Uther Mitchell Potter Anders Oreback Tucker Balch, Jim Bruce. Teambots. http://www-2.cs.cmu.edu/ trb/TeamBots/index.html, April 2000.

[WB01] Greg Welch and Gary Bishop. An introduction to the kalman filter. 2001.

[YFdL+01] T. Yamamoto, M. Fujita, M. de Lasa, S. Talebi, D. Jewell, R. Playter, and M. Raibert. Development of dynamic locomotion for the entertainment robot (teaching a new dog old tricks). *4th International Conference on Climbing and Walking Robots*, 2001.