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

Efficient Feature Detection Using RANSAC

By: Sean Harris

A Thesis submitted for the degree of
SOFTWARE ENGINEERING

August 24, 2011
Supervisor: Dr. Bernhard Hengst
Assessor: Dr. Maurice Pagnucco

Acknowledgements

I would like to thank my supervisor Bernhard Hengst for all his support and guidance throughout the year. He has been an amazing source of ideas and creativity and was always there to give me a boost along when I was lacking motivation.

I would also like to thank the rUNSWift team, it was an amazing experience to be part of a team with so many intelligent people where I was able to learn so much about design, programming and team work.

Finally I would like to thank my family and friends for all the support and encouragement you gave me throughout the year, especially when things got really busy.

Contents

Abstract

As a robot, trying to work out where you are on a soccer field is an extremely difficult task if you don't have many landmarks you can identify and use as a point of reference. This thesis presents the design and implementation of a field line detection system that uses a RANSAC based methodology to efficiently detect simple shapes like lines and circles. These simple shapes can then be combined to identify a variety of field line features around a soccer field including corners, T-intersections, the centre circle and the goal box. Each of these features can then in turn be used to calculate the position on a soccer field a robot is currently in.

Chapter 1

Introduction

1.1 Motivation

The motivation behind this work originates with the Robocup Standard Platform League Competition. This competition is focused around teams programming robots to compete autonomously in a game of soccer. One of the major difficulties with autonomously playing soccer is identifying landmarks on the field to help determine where the robot is currently located.

The main two identifiable landmarks on any standard soccer field are the goalposts at either end and the field lines across the playing area. Whilst the goalposts can provide a good estimate for a robot's location on the field, they are often a long way away and can be difficult to look at quickly and accurately. Field lines however, are present all around the field in a variety of shapes and formations, often in locations that are very convenient to look at and also provide accurate information about a robot's location. The aim of this thesis is to develop an efficient methodology to identify these field lines and the shapes they produce to help a robot localise on a soccer field.

1.2 Goals

The primary goals of this thesis are to develop a system with the following characteristics:

Accuracy: It should accurately report distances, headings and orientations of all identified field lines in compliance with a vision-localisation interface. There should also be few to no false positive identifications.

Speed: It should be fast enough that the entire perception pipeline can run on a 500Mghz processor in less than 30 milliseconds. This is because the camera gives out frames at a rate of 30fps, meaning that if the perception cycle runs slower than this, frames get missed. A faster perception cycle also improves the reaction time of the robot's behaviours, which is important in any sport.

Robustness: The system should be as noise resistant as possible to ensure that the quality of

information doesn't degrade and that the quantity of information is also high.

Modularity: It should be as modular as possible, to allow separate development on other aspects of vision by fellow members of the rUNSWift vision team.

1.3 Outline

The remainder of this thesis is organised into the following chapters:

Chapter 2: Background This chapter looks at the important background information relevent to this topic, including the rUNSWift system design, some other SPL teams' approaches to this problem and an introduction to the concept of RANSAC.

Chapter 3: Generating Candidate Points This chapter presents at the methodology used to determine points in an image that may be part of field lines, present the final results of the algorithm and reflect on its effectiveness.

Chatper 4: Detecting Simple Shapes: Lines and Circles This chapter discusses two methods for detecting lines and circles. The first and ultimately unsuccessful attempt is examined first as well as some reasons why it didn't work. The chapter then goes on to examine the final algorithm developed, present some results and analyse its efficiency.

Chapter 5: Detecting Field Features This chapter examines at the method for combining the simple shapes together to generate specific landmarks on the field. It also presents the results of the algorithm and discuss its effectiveness compared with the original plan.

Chapter 6: Speed This chapter presents the results of a variety of speed tests performed on the above algorithms on a robot under game conditions. It also then discusses the successes and shortcomings of the algorithms in terms of speed and efficiency.

Chapter 7: Future Work This chapter looks at some of the possibilities of extending the current system as well as different avenues of research that looked promising but weren't fully explored.

Chapter 8: Conclusion This chapter wraps up the thesis and concludes on the final results.

Chapter 2

Background

This chapter introduces the rUNSWift Vision Pipeline and some of the processes in the pipeline that impact Field Line Detection. It also examines the work of a variety of Robocup SPL teams as well as general computer vision algorithms relevant to this thesis.

2.1 rUNSWift Vision Pipeline

The rUNSWift Vision Pipeline [3] starts with a raw image from the camera. This raw frame, along with other information like a time stamp, colour classification, kinematics snapshot, etc, are stored and 3 saliency foveas are generated. Each saliency fovea is an 80x60 image containing either colour, grey scale or edge information depending on the type of saliency. This information is then passed down the pipeline to the first of the detectors, Field Edge Detection. After this each vision detector, including ball detection, field line detection, robot detection, etc. is run in sequence until they complete, at which point the cycle restarts with the next camera image. One of the important aspects of this pipeline is that information generated from any prior modules is available in the current module. For example field edge data is available for use in field line detection. This data pipeline is extremely important as most detectors rely on information generated by modules run earlier than themselves in the pipeline.

2.2 Edge Saliency

An important stage of the rUNSWift vision pipeline is the generation of the saliency images. There are three types of saliency images created:

Grey Scale Saliency The basis for generating the edge saliency image.

Edge Saliency The primary source of information for detecting field lines.

Colour Saliency Used to sanity check the results of many field line detection algorithms.

The Edge Saliency is an 80x60 image containing two values for each pixel. The first value represents the change across the horizontal direction (dx) and the second represents the change across the vertical direction (dy). It is generated by taking the grey scale saliency, applying a Gaussian smoothing operation over it and then calculating the dx and dy using a formula that models local change [3]. These two values are extremely powerful and allow us to determine both a magnitude and a direction for each of the pixels using simple formulas:

$$Magnitude = \sqrt{dx^2 + dy^2} \tag{2.1}$$

$$Direction = \arctan(\frac{dy}{dx}) \tag{2.2}$$

In practice, the first formula can be rearranged to avoid having to perform the extremely expensive square root operation, leaving us with:

$$Magnitude^2 = dx^2 + dy^2 \tag{2.3}$$

The magnitude and direction are extremely useful pieces of information and form the basis of the selection criteria for candidate points.

2.3 Image Plane vs Ground Plane

One of the important ideas in this thesis is the difference between the Image Plane and the Ground Plane when referring to an point's location. The difference is subtle however it has a large impact, especially on the different methodologies presented in Chapter 4.

Points that are in the Image Plane are represented in terms of their x and y coordinates in the saliency image. This means that a point in the very bottom right corner of the image is at (80, 60) whilst a point in the very top left of the image is at (0,0). In contrast to this, the Ground Plane represents the 2D plane of the physical floor space around the robot. So the point (0,0) in the ground plane is directly underneath the centre of the robot, whilst the point (1000, 0) would be 1 metre in front of the robot.

Pixels in the image have a direct mapping to points on the field and this is calculated through the kinematic chain developed by rUNSWift 2010 [10]. This tool is utilised in Chapters 4 and 5 where points and shapes are projected into the ground plane to make detecting shapes easier. To show the significance of the difference between the two planes, the image below shows a T-intersection where the lines in the image don't appear to intersect at 90 degrees. Once we project these points into the ground plane however, it is obvious that they actually do intersect at right angles.

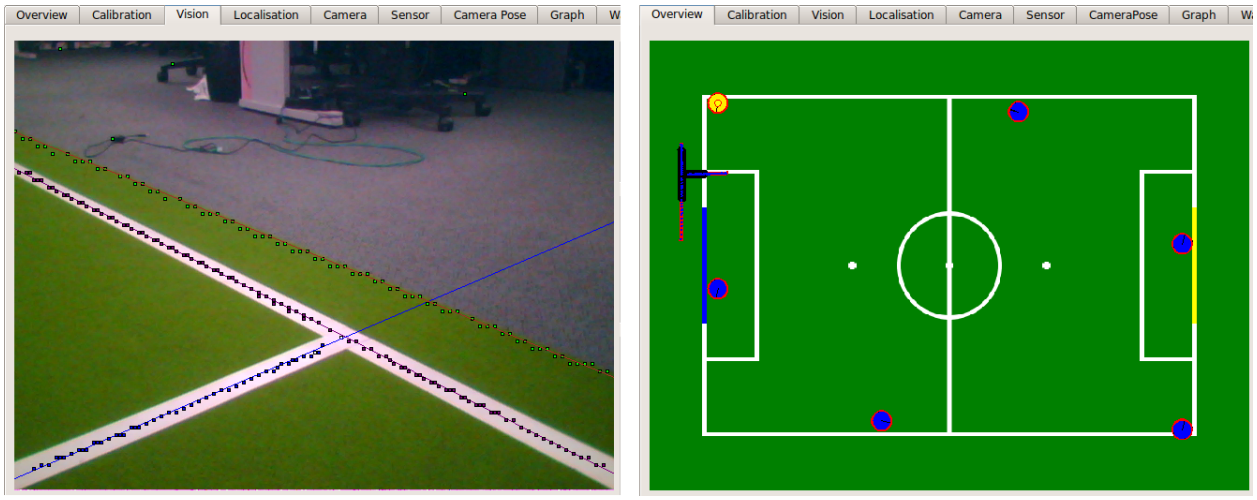

Figure 2.1: Lines in the vision plane (left) don't appear to intersect at right angles, yet in the ground plane (right) they actually do

2.4 Generating Candidate Points

The first step in most feature detection algorithms is to generate a list of candidate points or regions. This can be tackled in a variety of ways including analysing the colours of an image, running edge detection algorithms, using blob detection algorithms or even combinations of such methods. The following section will present a variety of methodologies for generating candidate points and regions used by other Robocup SPL teams.

In 2010 the rUNSWift team used colour as the basis for generating candidate field line points [10]. The algorithm involved scanning vertically down an image looking for changes in colour from green to white or white to green and marking each of these points as field line points.

This simplistic approach meant that this section of the algorithm ran fairly quickly, as there was little to no processing done on each pixel from the colour saliency. In addition to this, the rest of the vision pipeline also relied fairly heavily on colour, so the vertical scans were able to be optimised to also search for other features like robots or balls. This freed up valuable processing time for use at a later stage in the perception pipeline.

The speed and simplicity of the algorithm also has its downsides though. Using only a vertical scan also means that any field lines that are close to vertical in the image were difficult to detect. The reason for this is that the line might only appear in a very small number of the scans, which leads to very few candidate points being detected. The end result for rUNSWift 2010 was that any lines that were close to vertical were often missed or falsely detected as robots.

BHuman 2010 also used a similar approach of scanning vertically through a colour classified image [11], however they aimed to build up candidate regions as opposed to identifying candidate points. Their algorithm aimed to use the vertical scan to identify small regions and then group adjacent

regions together to form larger ones. If not enough similar regions were found nearby, small regions were discarded as false positives. The vertical nature of their scan meant that they too had to deal with the issue of vertical lines being difficult to detect, especially since such lines would appear as only a small number of regions. As a solution the team added a special case to consider how tall the region was, so that vertical field lines weren't missed.

Nao Devils 2010 also took a similar approach of scanning through a colour classified image [5], however they used radial scan lines instead of vertical scan lines. Radial scan lines originate at the centre of the robot and pan outwards like the radii of a circle. The effect this gave on the image scan lines is that they were almost vertical, but ever so slightly tilted. Thus the team again had to deal with the problem of vertical lines not being detected by the scans. The solution Nao Devils 2010 used was to do a small number of horizontal scans on sections of the image they thought were important to try and pick up candidate points they had missed.

Another factor of each of the previously mentioned approaches is that they are all dependent on accurate colour calibrations. While this problem can be avoided by having a good colour classification, it can sometimes be difficult to imitate the exact lighting conditions of a game situation. Often if a large crowd gather around a game or the sun suddenly shines brighter through a window, the lighting conditions change enough that the colour calibration becomes less reliable and errors can propagate through the entire vision pipeline. It is also a very time consuming solution because calibrating colours perfectly takes an extremely long time. Thus having a heavy dependency on these colour tables is a risky approach.

Nao Team HTWK from Leipzig in Germany has a unique approach for their vision algorithms where they use almost no colour classification (aside from a small amount for the goalposts) [12]. They instead rely heavily on edge information to generate candidate field line points as well as most other aspects of their vision data. This means that they aren't nearly as vulnerable to changes in lighting conditions and don't need to spend lots of time setting up colour tables in each new environment.

As you can see from the above, present methods for detecting candidate field line points rarely work in all cases. Most teams are required to implement extra functionality to accurately identify points on vertical field lines and many are also heavily reliant on colour tables.

2.5 RANSAC Based Algorithms for Image Analysis

Random Sample Consensus (RANSAC) is an iterative approach to estimate parameters of a model from a set of data that contains outliers. The algorithm works on the assumption that the data consists of inliers, whose distribution can be explained by some set of parameters, and outliers who don't fit the model. A relevant example is fitting a line to a set of points in 2D space and is demonstrated in Figure 2.2.

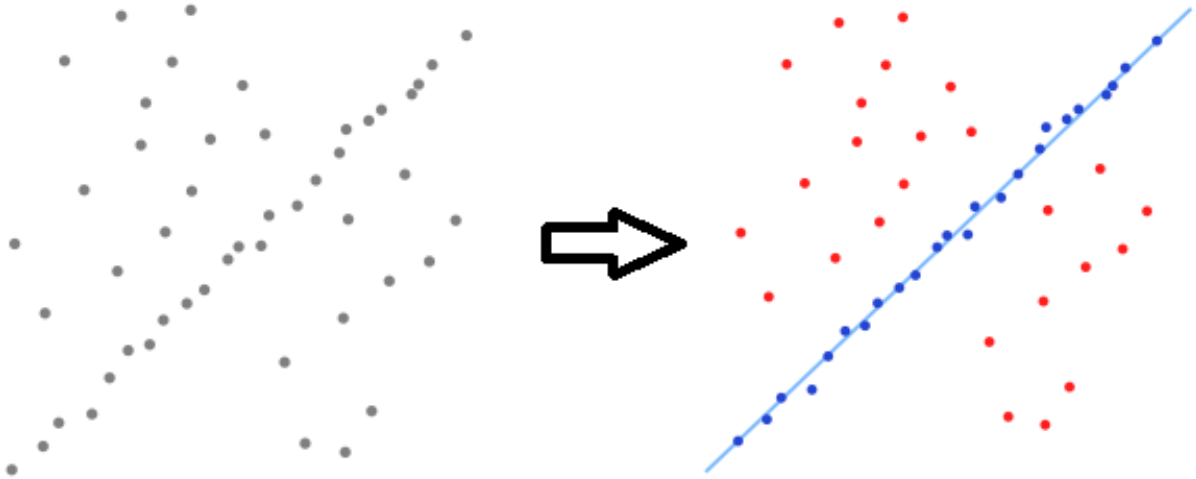

Figure 2.2: A data set with many outliers having a line fitted by the RANSAC algorithm. Notice that the line best fits the inliers instead of the entire data set

Starting with a set of points in 2D space, the algorithm picks two points and generates a line between them. It then calculates which points from the original data set are now considered part of this line based on their distance to the line. Each of these 'inliers' are now considered part of the line and are used to generate some sort of evaluation of the line's fit through the dataset. This is repeated for a predetermined number of runs, during which time it is hoped that two points from the correct subset of data are chosen. The algorithm works well because the line fits the set of inliers optimally instead of trying to fit the entire dataset optimally. Parameters for the function include:

n : the minimum number of points required for a line (2).

k : the number of iterations.

t : a distance measure of how far a point can be to still be considered as part of a line.

d : the minimum number of points required for the line to be a good fit.

While this is a fairly specific example, the above methodology can be used on a variety of shapes (including circles) as well as applications not relevant to computer vision algorithms. It is a useful algorithm because of its good speed versus accuracy trade off where only a very small amount of accuracy is traded away (especially if the data set contains lots of inliers) for a very large increase in speed.

2.6 Detecting Simple Shapes

Once a list of candidate points has been generated, the next step is to try and form lines and circles using these points. This section presents a variety of methods used by other Robocup SPL teams

as well as some more traditional computer vision algorithms used to find lines and circles.

In 2010 BHuman used a clustering based algorithm similar to the quality threshold algorithm in [8] to group small segments they had generated into long field lines [11]. Their method included representing each segment in Hesse normal form then grouping segments that have similar distance and direction values into longer lines. The problem with the clustering algorithm was that it had been modified to run in $O(n^2)$ time worst case and thus wasn't guaranteed to find optimal clusters. Despite this, the team claimed that the speed and accuracy of the algorithm were still acceptable for its use.

To detect circles the team used all the line segments that were too small or ruled out when trying to make straight lines. For each of the left over lines, they ran through every pair of segments on the line to try and detect if the segments formed a circular shape. The method of doing this was to calculate the perpendicular from the middle of the segment and intersect it with the perpendicular from the middle of the adjacent segment and check the distance to that point. If that distance was similar to that of the centre circle then it was assumed to be part of the centre circle and so a spot was generated to represent what should be the centre of the circle. Each of these segments was then clustered using the previously mentioned clustering algorithm.

The biggest problem with a clustering approach was that to ensure it ran in an acceptable time frame, optimal clusters were sacrificed. This meant that there were lots of cases where small segments are excluded or left out and the algorithm required a large number of different sanity checks to ensure that each line was actually a full line. Whilst this approach can be extremely messy, BHuman have shown that it can still be made to work.

The Nao Devils 2010 used a chaining algorithm to group candidate points into chains that represent line or circle segments [5]. This chaining ran in linear time and linked points based on their distance and order in the list. Once this was completed they were left with lots of segments, some of which belonged to the same line / circle and some of which that didn't. Thus they were in a similar situation as BHuman 2010 and then had to merge similar line segments. The Nao Devils had a more efficient way of doing this though as they were able to work out the parameters of each chain and extend it to include nearby chains with similar values, forming neat lines. Despite the efficiency improvements here however, there were still 2 steps in this phase and a lot of sanity checks needed to ensure false positives didn't occur.

The Nao Devils' circle detection algorithm also used the above chains and simply looked at each chain to see if there was a trend of directional change along the chain at a curvature roughly equal to the centre circle. This meant that segments were classified as either line or circle segments before any merging occurs. One big advantage of this was that the centre circle could be identified even when only a small fraction of the circle was visible. This was an extremely useful feature since robots very rarely get perfect frames in a game situation and need to make the most of every glimpse of a feature.

rUNSWift 2010 however, used none of the above methods [10]. In fact, rUNSWift 2010 didn't

attempt to detect any sort of shapes out of field lines at all. Instead the team simply relied on the candidate field line points and the current localisation position to match the field of points onto a nearby section of field line area. Whilst this approach was useful for fine tuning a localised robot's exact position, if the robot was lost it was almost entirely useless as no actual features were detected.

One interesting section of the rUNSWift 2010 code base was the algorithm used to detect the edge of the field. There was a candidate point selection part of the algorithm that used the first green pixel in a vertical scan to represent a possible start point on the field and then a RANSAC based approach to finding lines of best fit for those points [7]. This RANSAC approach involved picking two points randomly, drawing a line through them and calculating how good a fit it was for the data set. This was repeated a constant number of times with the best line being kept as the final line. The end result was one or two lines of best fit (depending on the frame) that matched the edge of the field. This approach was not only fast and simple, but also managed to be fairly accurate, especially considering the dataset was extremely noisy at times. This methodology forms the basis for the next part of our algorithm.

UPenn 2010 took a unique approach and optimised the otherwise too slow Hough transform [6] for use in line detection [2]. They used the Hough transform's intersection of sinusoidal curves in polar space to generate an array of discretised values that represent where lines might have been located. The paper doesn't provide a high level of detail about just how fast the algorithm ran, but it is suspected that it still carried a significant overhead in comparison to other line detection methodologies. The team also only used the global maxima from the values collected (possibly due to speed restraints) so doesn't really test the full capabilities of the Hough transform to detect multiple lines and circles in an image.

Thus it is clear that the rUNSWift code base is a long way behind many of the advanced teams by not attempting to detect even simple shapes in images. There is also a large scope for innovation within the Robocup SPL community for utilising efficient and accurate line and circle detection algorithms.

2.7 Detecting Field Features

The final step is to now combine all the information so far to form useful features to pass onto localisation. This section examines some of the possible landmarks that exist on the field and some approaches used by other SPL teams.

The Robocup Standard Platform League (SPL) uses a standard field format each year for competition. This means that it is known in advanced exactly what the field will look like and what features each team can design the system to look for when playing. In 2011 the field layout looked as follows:

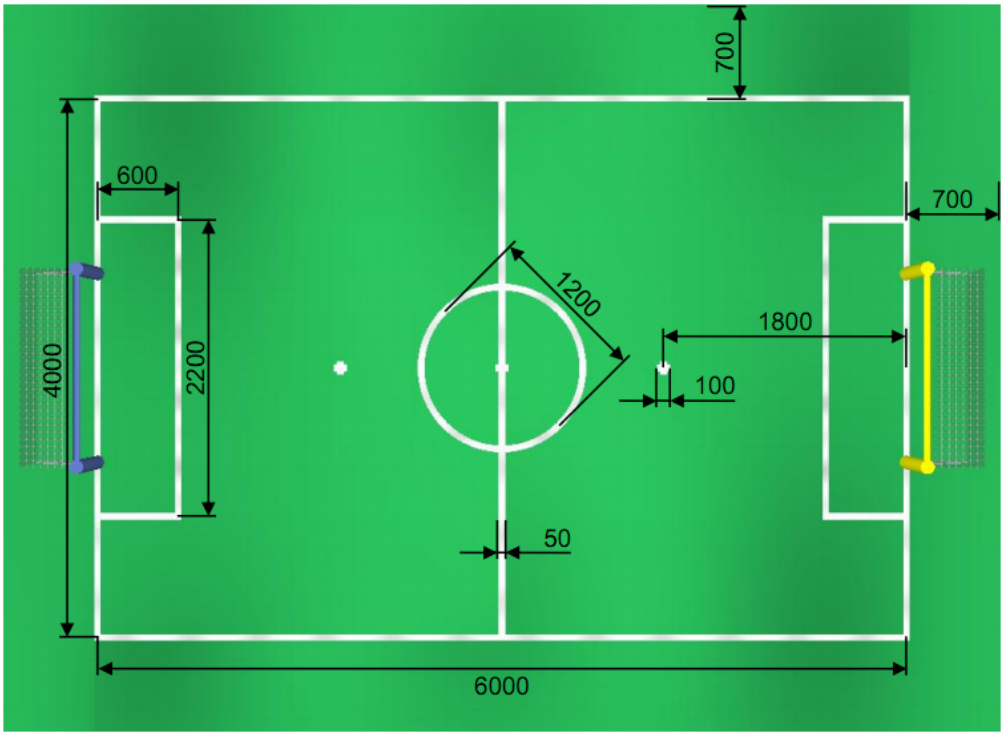

Figure 2.3: The standard field layout for Robocup SPL 2011

As you can see it looks similar to a normal soccer field and thsu has a number of landmarks that can be identified. Some of the notable landmarks include:

Lines : There are 11 straight lines spread around the field.

Corners : Two field lines intersect in 8 places on the field to form a corner.

T-Intersection : Two field lines intersect in 6 places on the field to form a T-intersection.

Centre Circle : The circle in the middle of the field is the only circle and has a line intersecting it.

Penalty Spot : There are two penalty spots on the field, one in each half.

Goal Box : The goal area is surrounded by a box shape, which can be broken down into two pairs of parallel lines.

All of these features have useful information to pass through to localisation and can be vital in keeping the robot localised without having to look around for goal posts. However there are a lot of features listed there and few teams can actually detect all of them. Here is some discussion about what some of the 2010 teams thought were useful.

The team from UPenn were only able to detect a single line per frame [2]. Unfortunately this ruled out detecting any complicated features, meaning that their field features were only able to be used

to assist the fine tuning of a current position. Whilst this is still very useful for maintaining a good localisation, it isn't very useful if the robot becomes lost or gets kidnapped.

UTAustin 2010 were able to utilise lines and circles in each frame [1]. The use of multiple lines instead of just a single line like UPenn gave them greater precision when calculating a position from the lines. It also allowed for some improved kidnapped robot recovery if multiple lines can be matched to sections on the field. The addition of the centre circle detection was also a big leg up as it had the power to provide just two symmetrical hypotheses on the field with a good enough observation, although UTAustin don't give enough detail to determine if they were capable of actually refining the observation enough to do this.

The Edinferno 2011 team have an interesting combination where they detect the centre circle and the penalty spot [9]. This gives them a fairly good distribution of landmarks across the field however they are missing a huge resource by not considering any straight lines. Straight lines are by far the most prevalent landmark and although this makes it less useful for a kidnapped robot, it can be extremely useful for fine tuning the current position of a fairly well localised robot.

The Nao Devils 2010 team were one of the best at generating landmarks from field lines [5]. They detected lines, corners, T-intersections, unidentified intersections and the centre circle. The corners and t-intersections were extremely useful for both fine tuning a well localised robot's position and for the kidnapped robot problem. The unidentified intersection was a great method of utilising every piece of information in the frame, even if the frame doesn't allow for detection of exactly what sort of intersection had been seen, the data was still used. The Nao Devils 2010 also utilised the full power of the centre circle by using the line going through the centre of it. The combination of the line and circle allowed just 2 symmetrical hypotheses to be generated and made it an extremely useful update both for a well localised robot and for a kidnapped robot.

Although these teams are all making good use of field features, many Robocup SPL teams don't actually detect even the simplest of shapes. rUNSWift 2010 didn't detect any sort of shapes and thus lost all ability for field lines to be useful for a lost robot [10]. They were able to do a local update to tune a localised robot's position using just a set of points, however this approach is useless if the robot isn't close to where it suspects it is. Thus this is an area where both the rUNSWift team and the Standard Platform League as a whole have a large knowledge gap and the aim of this thesis is to research and develop a system to fill this gap. The rest of this report will explain the methodology and reflect on the results of this research and development.

Chapter 3

Generating Candidate Points

The first step in the process of detecting field features is to identify candidate points that may make up part of each feature. The general process for doing this involves some sort of scan through the image that detects points based on colours or edges present. In this chapter we will look at the solution developed here, compare it to some other solutions and discuss its overall effectiveness.

3.1 Methodology

This approach is focused around utilizing edge data to find good candidate points. The basic principle is to scan vertically and horizontally through the image, looking for a pair of edges that fulfil the criteria of possible field line points. The idea behind this is to reduce the dependency on the colour saliency, which is extremely vulnerable to lighting conditions and remove the need for a special case later down the track to detect vertical lines. The algorithm is shown in Algorithm 1 and we will step through it here to show how it works.

The first step is to scan vertically through the edge saliency image. Here field edges are utilised to reduce the area that needs to be searched to generate candidate field line points. If field edges are detected in the image then the algorithm only scans below the field edge, since field lines are never found above the field edge anyway. If no field edge is found however, the entire image must be scanned. The result of this is that if a field edge is detected, instead of starting the vertical scan at the top, we start at the field edge and work our way down.

For each pixel the first thing to consider is how strong the magnitude of edge is at that point. If the edge is weak, we simply skip it and move on. If it is a strong edge though, we need to consider which side of the field line it might represent (since it could be a green - white edge or a white - green edge).

It is a reasonable assumption to make that the first edge found should be a green - white edge as there should been some green at the edge of the field before the line starts. Once this first edge is found, we store its magnitude and direction for use later and move to the next point.

Algorithm 1 Generating Candidate Points

```
for all i ∈ columns do
  top ← (0, 0)
  bottom ← (0, 0)
  for all j ∈ rows do
    magnitude^2 ← edges[i][j].x^2 + edges[i][j].y^2
    if magnitude^2 > strongEdgeThreshold then
      if top = 0 then
        top ← edges[i][j]
      else if similarDirection(top, edges[i][j]) then
        if magnitude^2 > topMagnitude^2 then
          top ← edges[i][j]
        end if
      else if bottom = 0 then
        if oppositeDirection(top, edges[i][j] then
          bottom ← edges[i][j]
        end if
      else if similarDirection(bottom, edges[i][j]) then
        if magnitude^2 > topMagnitude^2 then
          top ← edges[i][j]
        end if
      end if
    else if top ≠ 0 AND bottom ≠ 0 then
      midpoint ← (top+bottom)/2
      if colour[midpoint] ≠ white then
        top ← (0, 0)
        bottom ← (0, 0)
        continue
      end if
      if distance(top, bottom) > widthThreshold then
        top ← (0, 0)
        bottom ← (0, 0)
        continue
      end if
      if distance(0, midpoint) > distanceThreshold then
        top ← (0, 0)
        bottom ← (0, 0)
        continue
      end if
      points.add(midpoint)
    end if
  end for
end for
```

Upon reaching the next strong edge point, we need to consider whether it is part of the first edge, part of the matching white-green edge on the other side of the field line, or just a random bit of noise. The way this is done is by examining the direction of the edge. If the edge is a similar direction to the first edge, it is probably part of that top edge and has just spanned more than 1 pixel in width. In this case, we compare the magnitudes of this point and the currently saved top edge point and only keep the strongest pixel for that top edge.

If the next point's direction had instead been roughly opposite to the first point, then it would be considered as part of the bottom side of the edge. Similar to the top side, there can be multiple points detected as part of the bottom edge (assuming they all have a similar direction), but only the point with the highest magnitude is stored.

There is also a possibility that the second point fits neither of the above two categories, in which case it is assumed that the first point mustn't have actually been the top of a field line and thus is considered noise. Since we can't tell if the second point is valid or not yet, we set it to be the new first point and continue the search for a second point.

The scan continues downwards until either it hits the bottom, or it stops hitting strong edges. If it stops hitting strong edges then we have probably finished scanning over the field line so we stop and process the information we have so far. If we have detected both a top edge and a matching bottom edge then we calculate the midpoint of the two and in theory this should give us the centre of the field line. To confirm the hypothesis we need perform a variety of sanity checks on the information to ensure that we have actually identified a good candidate point. The sanity checks for each point include:

- The top and bottom points should be opposite in direction (already tested implicitly by the algorithm)
- The top and bottom points should be roughly 50mm apart
- The midpoint shouldn't be too far away
- The midpoint should be white in the Colour Saliency

If all the sanity checks are passed, then we consider the point to be a good candidate for a field line point and save it in a list along with some information about the point. We store a dx and dy value for the midpoint, which is calculated as the average of the top and bottom point's dx and dy, the coordinates of the pixel in the image plane as well as the coordinates of the pixel in the ground plane.

Once all the columns have been scanned, the next step is to determine if any horizontal scans are necessary. We don't want to scan over areas that have already been scanned because it wastes time and resources and generates duplicates in our list of points. Thus the horizontal scan is only performed if the row has some section below the field edge and if the row doesn't already have some field line points in it.

If we decide to perform a horizontal scan, the algorithm used is the same as the vertical scan, except that the top and bottom of the line now represent the left and the right sides of the line. None of the actual logic for finding points changes.

3.2 Results

To show the results of the algorithm, the robot was placed in a variety of realistic localisations on the field. The rUNSWift debugging tool Offnao has a vision tab which is used to display what the robot can see at any one moment as well as the results of the candidate point selection algorithm. All the pictures below are screenshots taken of the Offnao tab for some scenarios set up to show the full capabilities and limitations of the algorithm.

3.2.1 Experiment 1 - Centre Circle

The first experiment involves placing a robot near the centre circle looking towards the area where the circle intersects the halfway line. Here in Figure 3.1 we've added some extra display information to show the top and bottom edge points of the field lines to show how the pair matching works. The top/left edges are represented by the red pixels, the bottom/right edges are represented by the blue pixels and the midpoints are represented by white pixels. When the top and bottom edges are adjacent the midpoint has the same pixel location as one of the edges, thus gets drawn over in some occasions. Here we can see that the algorithm has picked up a good number of points both on the circle and on the line intersecting it.

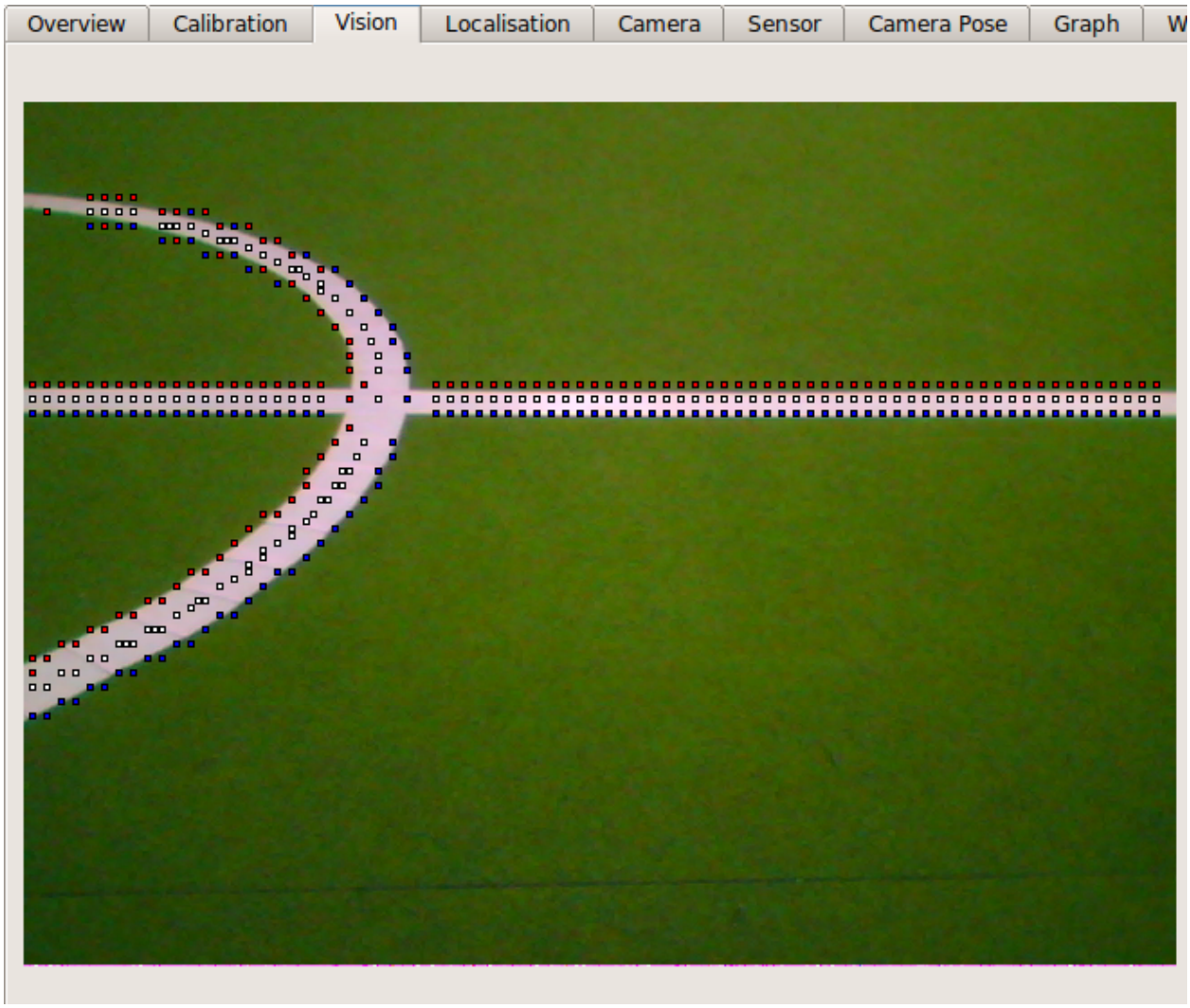

Figure 3.1: Candidate point selection algorithm shown with red representing the top edges, white the midpoints and blue the bottom edges

3.2.2 Experiment 2 - T-Intersection

The second experiment involves placing a robot near the halfway line and facing it towards the T-Intersection the halfway line forms with the sideline. Figure 3.2 shows the raw image and the edge saliency image, both with the top, bottom and middle pixels overlayed like the previous experiment. In this picture some of the top and bottom pixels have painted over each other with the horizontal and vertical scans, so it isn't always possible to find the matching edge in a pair.

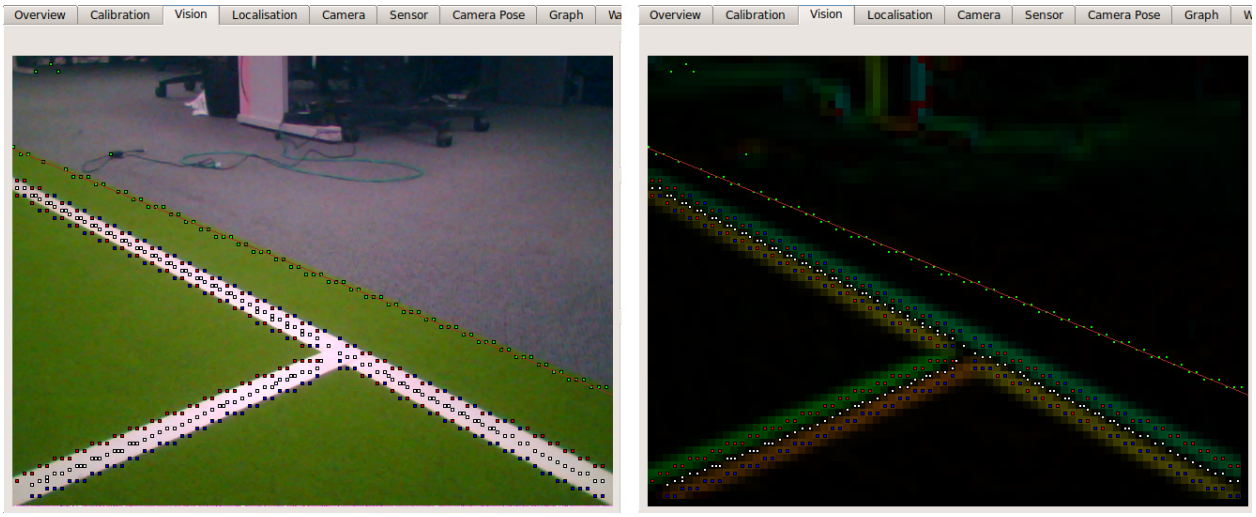

Figure 3.2: Candidate point selection algorithm shown with red representing the top edges, white the midpoints and blue the bottom edges

3.2.3 Experiment 3 - Goal Line

The third experiment involves placing the robot about 1m from the goal line, near the left hand post of the blue goals. The yellow circle in Figure 3.3 shows a rough location for the robot. As you can see the vertical scan has picked up a good set of points on the goal line and the top of the goal box, which are shown in by the purple and red points respectively, while the horizontal scan has picked up a good set of points on the vertical goal box line, shown by the orange points.

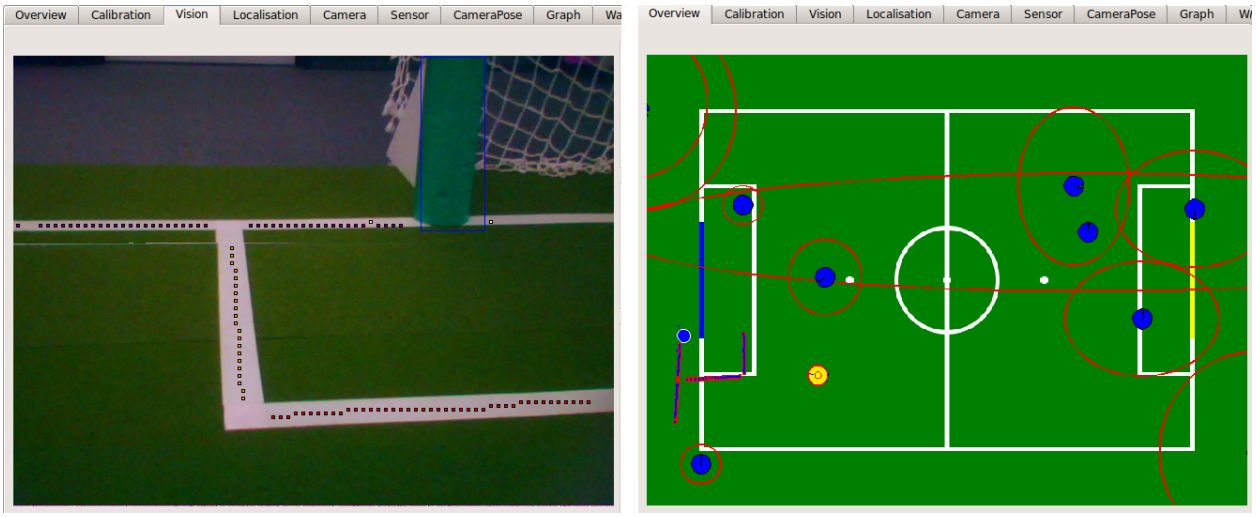

Figure 3.3: Candidate point selection algorithm showing the points being detected by the vertical scan (purple / red) and the horizontal scan (orange)

3.2.4 Experiment 4 - Robots

The fourth experiment involves placing a robot about a metre in front of the active robot to test the number of false positive points generated by a robot in the frame. Note that this is best performed near the penalty spot area due to the lack of actual field lines present in the frame, but can essentially be run anywhere on a field. This is one of the worst possible cases for false positives and as you can see in Figure 3.4 quite a large number of candidate points have been detected inside the robot, especially its arms. It is also worth pointing out that nearly all of these points were generated by the horizontal scan.

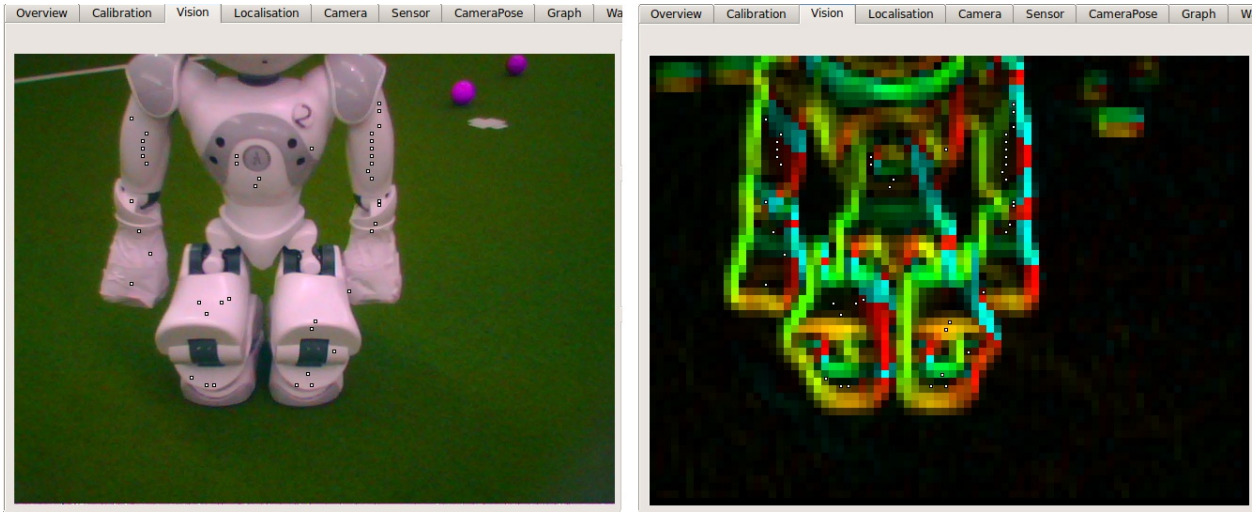

Figure 3.4: Candidate point selection algorithm incorrectly detecting candidate points inside a robot

3.2.5 Experiment 5 - Maximum Distances

The fifth experiment involves placing the robot in a similar position to experiment 3, but another 0.5m further away, as shown by the yellow circle in Figure 3.5. In this scenario the goal line is approximately 1.5m away and very few candidate points are generated on it. The other two lines on the goal box however both generate a fair set of candidate points via the horizontal and vertical scans.

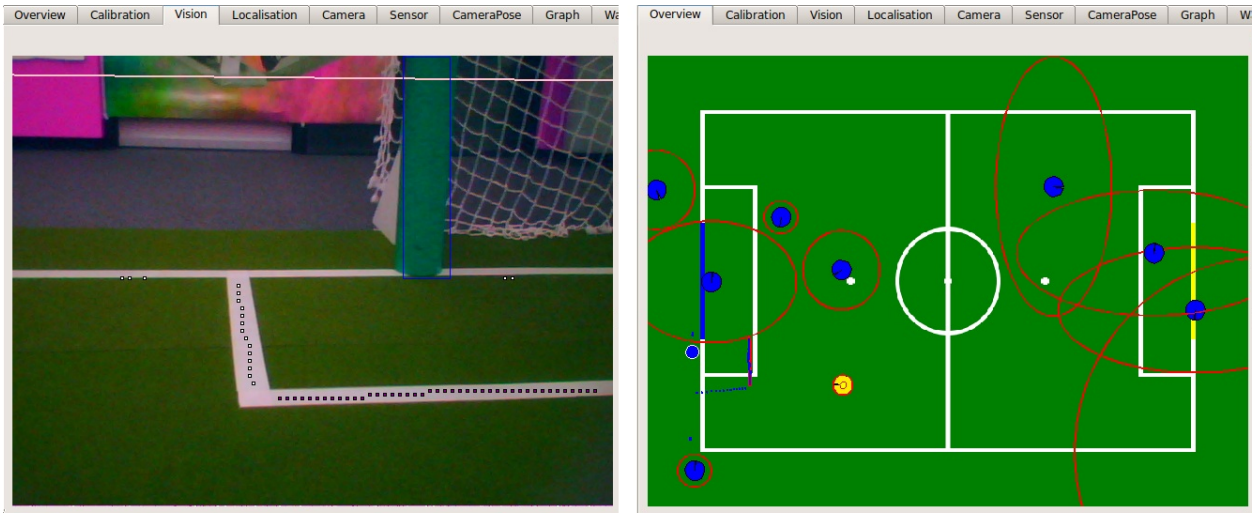

Figure 3.5: Candidate point selection algorithm struggles to identify the goal line at a distance of around 1.5m

3.2.6 Metrics

There were quite a few metrics used in this section of the algorithm. Here is a list each of their values and an explanation of the reasoning behind each one.

Minimum magnitude to be considered a strong edge: 250

This value was a chosen because it excluded all of the blank sections of the field that were just green but didn't exclude any strong edges. This value was made lower rather than higher to ensure that it didn't rule out any actual field line edges extremely early on in the algorithm.

Maximum difference between the absolute values of the top bottom edge angles: 35 degrees

This threshold was also kept quite loose to ensure minimal good data was excluded due to small amount of noise in the edge saliency. It was still well and truly tight enough to remove most of the noisy points detected and rarely removed any actual field line points.

Maximum distance from top edge to bottom edge: 200mm

This value is a trade-off between detecting far away field lines, where the top and bottom edges appear to be a long way apart, and removing robot points that clearly have the two edges a long distance apart. This value was chosen to maximise the chances of detecting far away field lines, whilst attempting to keep the number of robot points below the threshold where they may get detected as part of a field feature.

Minimum distance from top edge to bottom edge: 40mm

This minimum distance may seem like a very tight threshold especially considering the value of the maximum distance, however there was very rarely any actual field line points excluded by this

metric. The reason for this is that in pixels more than a metre away neighbouring pixels can be more than 40mm apart, meaning this test can't possibly fail. In pixels close to the robot however, the accuracy of the distance measurements is quite high meaning we can afford to be stricter on the minimum threshold.

Maximum distance away that midpoint can be: 3000mm

Since actual field line points were rarely detected at long distances, any points that were further away than 3m were most likely part of robots, so should be excluded. This was left as a fairly loose metric to ensure that no correct field line points were discarded by it.

3.3 Evaluation

From the above results it is clear that field line points can be accurately detected up to a distance of 1.5 metres away. Any field lines that appear in the image and are closer than this distance show up very nicely and generate a large set of points that are easily matched to a shape later down the track.

For lines that appear further than this 1.5 metre mark however, there are few to no points detected. This deterioration is primarily due to inaccuracies in distance measurements and colour classifications that result from having such a low resolution saliency image. As pixels in the image get further and further away, each one represents an increasingly larger area in the ground plane. This makes the colour classification less accurate as it now has to classify a pixel that might be half white and half green, as either one or the other. This means that by the time you start to get around the 2m mark, the white pixels for the field lines don't consistently show up in the colour saliency because each pixel has so much green in it. The result of this is that sometimes valid points get detected, but because the colour saliency says the pixel is actually green, they get thrown away.

The other issue cause by the low resolution is that the distance measurements become far less accurate as objects get further away. At a distance of around 1.5 metres, vertically adjacent pixels are around 50mm apart. This means that the top and bottom pixels need to be literally adjacent to be approximately 50mm apart. To compensate for these inaccuracies, the sanity check for the top and bottom pixels being approximately 50mm apart actually allows them to be up to 200mm apart before rejecting them. Unfortunately in the edge saliency, field lines at a distance of about 1.5m can have the top and bottom edges being 4-5 pixels apart, which translate into 250mm in the ground plane. Thus this sanity check often eliminates far away field line points simply because of inaccurate distance measurements.

Despite it often ruling out good data simply because it is far away, this sanity check is actually extremely useful at eliminating points detected inside robots. Being that robots are a similar colour to the field lines, they generate a very similar edge in the grey scale and hence the edge saliency image and also appear as white pixels in the colour saliency scan. This makes them very difficult to distinguish from real field line points, which is why the width sanity check was introduced. Robots

are much wider than field lines are, so having a check for the distance between the top and bottom edges allows us to eliminate many robot points. In saying that, not all robot points can be nicely eliminated using this method. Robot arms in particular are about the same width as a field line, which means that in bad frames they can generate around 6-8 field line points each. This number could be reduced by tightening the width sanity check threshold down from 200mm to 100mm, but then more actual field line points get lost due to inaccuracies with distance measurements as objects get further away. Thus there is a trade-off here where we are forced to allow some robot points to be included for us to be able to detect field line points up to 1.5m away.

Chapter 4

Detecting Simple Shapes: Lines and Circles

The next step in the process of detecting features is to first detect simples shapes that can be combined into larger landmarks later on. The simple features that we aim to detect here are lines and circles.

4.1 Methodology

Before looking at the final solution for this section, it's worth reviewing an earlier avenue of research because it looked extremely promising is many situations. The original aim was to remove the workload from the RANSAC section by pre-grouping the points, but this had a few crucial flaws and was eventually abandoned.

4.1.1 Pre-grouping Points Based on Edge Information

One of the early algorithms for finding simple shapes involved using the edge direction of each of the candidate points to group them into separate lists. The algorithm relied on the fact that points on a straight line should all have a very similar edge direction if they all belong to the same line (or another parallel line). Based on this principle, points were sorted into buckets using the follow algorithm:

Algorithm 2 Bucketing Candidate Field Line Points

for all $p \in points$ **do**
  $binNumber \leftarrow \frac{p.angle+\pi}{binSize}$
end for

Once separated out into discrete groups, the next step was to look through the bins and find the window of n bins that had the most points in it. Once the best window was found, those points

were all stored in a separate list to use later and their corresponding buckets were reset to zero so the algorithm could run again. This loop was repeated until there weren't any more full buckets and the leftover points were put into a leftovers list and stored with the other lists. The algorithm for picking the best window of points was as follows:

Algorithm 3 Finding the Best Group of Points

$j \leftarrow 0$
while $j < windowSize$ **do**
  $windowSum \leftarrow windowSum + bin[j]$
  $j \leftarrow j + 1$
end while
$bestSum \leftarrow windowSum$
$i \leftarrow 1$
while $i < numberBins$ **do**
  $windowSum \leftarrow windowSum - bin[i-1]$
  $windowSum \leftarrow windowSum + bin[(i + windowSize - 1)\%numberBins]$
  if $windowSum > bestSum$ **then**
    $bestSum \leftarrow windowSum$
  end if
end while

Once the candidate points had been grouped based on their edge values, the next step was to run a RANSAC line detector on each list. The points in the list were all represented in image space (ie, in terms of their pixel x and y values), not in the ground plane. The RANSAC line detector needed to be run twice on each list since parallel lines have the same direction and were be grouped into the same list. The RANSAC line algorithm ran as follows:

Algorithm 4 RANSAC Line

$i \leftarrow 0$
$bestVariance \leftarrow max$
while $i < k$ **do**
  $p1 = points[rand\%size]$
  $p2 = points[rand\%size]$
  $line = line(p1, p2)$
  for all $p \in points$ **do**
    $distance = \frac{|line.t1 * p.x + line.t2 * p.y + line.t3|}{\sqrt{line.t1^2 + line.t2^2}}$
    if $distance < e$ **then**
      $variance \leftarrow variance + distance$
      $numPoints \leftarrow numPoints + 1$
    end if
  end for
  $variance \leftarrow variance * 0.2 - numPoints$
  if $variance < bestVariance$ **then**
    $bestVariance = variance$
  end if
  $i \leftarrow i + 1$
end while

The variance for this algorithm was based on the sum of the distances to each point considered on the line as well as the number of points that were considered on the line. More points and a smaller distance sum contributed to a lower variance and thus a better fitting line for the dataset.

If the RANSAC line algorithm succeeded in finding a line with a low enough variance then all the points considered part of that line were removed from the list and if there were still enough points then the algorithm was run again. This was repeated for each list and any leftover points were collated together for circle detection afterwards.

In theory, at this stage there should have been a set of lines and their corresponding points as well as a list of leftover points. If there was a circle also present in the frame, the RANSAC line function shouldn't have matched it and thus we should have now been able to run a circle based RANSAC algorithm to find if the centre circle was also present.

The RANSAC circle algorithm works in a similar fashion to the RANSAC line algorithm, although has some slight variations. It runs as follows:

Algorithm 5 RANSAC Circle

$i \leftarrow 0$
$radius = 600$
$bestVariance \leftarrow max$
while $i < k$ **do**
  $p1 = points[rand\%size]$
  $p2 = points[rand\%size]$
  $circle = circle(p1, p2, radius)$
  for all $p \in points$ **do**
    $c \leftarrow circle.centre$
    $distance = \sqrt{(c.x - p.x)^2 + (c.y - p.y)^2} - radius$
    if $distance < e$ **then**
      $variance \leftarrow variance + distance^2$
      $numPoints \leftarrow numPoints + 1$
    end if
  end for
  $variance \leftarrow variance * 0.2 - numPoints$
  if $variance < bestVariance$ **then**
    $bestVariance = variance$
  end if
  $i \leftarrow i + 1$
end while

Once this had run all the lines and circles in the image should have been successfully identified.

Results and Issues

This binning concept worked extremely well when analysing straight lines. Since all the straight lines on a soccer field are perpendicular, the end result was either one or two (depending on the frame) clearly defined bins which represented the different directions of the lines in the frame. The

two pictures in Figure 4.1 show the points being grouped into a blue group and a purple group both of which are perpendicular to each other. These two groups were then easily matched to a line using RANSAC because there were essentially no outliers in any of the sets of points.

Figure 4.1: Pre-grouping points results in two groups, blue points and purple points, which are easily matched to a line using RANSAC

One of the useful things about this approach was that the RANSAC line algorithm had very little work to do on the grouped sets, because they only contain up to two lines, both of which are parallel and usually easy to distinguish from each other. It was also a fast algorithm and maked it easy to exclude robot points because they often didn't fall within one of the large bins.

Unfortunately this approach had 1 major flaw; the binning concept didn't work for circles. The basis of this approach was that all the points on a line would have a similar edge direction, and whilst this is true for lines, the points around a circle don't fit that model. The directions of the circle points change with the curvature of the circle so these points don't fit into any nice bin like points on the straight lines do. Instead short segments of the circle were grouped together and turned into short lines. Some examples of this are shown in Figure 4.2.

Figure 4.2: Pregrouping points generating a large number of short line segments from the centre circle

Figure 4.3: Pregrouping points not able to distinguish circle segments from line segments

As you can see, the halfway line was binned very nicely, but the points around the circle didn't group particularly well. As shown in Figure 4.3, the circle points often ended up being binned into small groups and got detected as short line segments instead of circle segments. This approach was worked on for a fairly substantial amount of time, but the issue of circle detection wasn't able to be fixed, so it was eventually abandoned in favour of the method presented in the next section. It is included here because it had so many promising features to it and hopefully can be utilised in the future.

4.1.2 Simultaneous RANSAC Lines and Circles

There are three significant differences between the final solution used and the one proposed above. First of all, since the bucketing didn't work well for circle points, any attempt to pre-sort candidate

points has been removed. Unfortunately this means that we lose a lot of the neat features provided by the edge data to remove the workload from the RANSAC algorithm. The second change is to map all of the points into the ground plane, rather than analyse them in image space (see Section 2.3 for more information on Image Plane vs Ground Plane). The idea is that this would help distinguish curves from straight lines.

The final and most significant change is to combine the RANSAC line and circle functions into one. The concept for this came from the problem where the RANSAC line would pick up some pieces better suited to the circle, whilst the RANSAC circle would pick up some pieces better suited to a line. The idea is to essentially run the two algorithms in parallel, compare the final results and take the better of the two. The modified RANSAC algorithm runs as shown in Algorithm 6. The end result is that we end up with either a line or a circle, whichever is better suited to the set of points.

One of the challenges with this approach is that it requires a good metric for comparing circles and lines to each other. Whilst the variance of a line is a good comparison against other lines, whether or not it could be used as a comparison against circles as well was a big uncertainty. There are also a variety of different methods for calculating the variance of a shape that can be used.

In the end a fairly simple variance calculation involving the sum of the distances to each point and the number of points, with a slight bias towards lines, was enough to accurately compare the two shapes and ensure that even a small segment of either shape was accurately identified as the correct shape. The final top level algorithm for finding lines and circles is shown in Algorithm 7.

The only sanity check for the shapes detected is a check to see if a line has been extended too far. The line at the top of the goal box can sometimes get extended out to include a few points on the sideline as well, which is incorrect and causes features down the line to be incorrectly identified. This function simply involves scanning along the first few pixels of each line detected and ensuring that we don't get lots of green there. If we do in fact find lots of green here, we just need to keep tracking across the line until we hit a white pixel and we can set that as the new end point.

4.2 Results

To show the results of the RANSAC line and circle algorithm, the robot was placed in a variety of realistic localisations on the field and the output recorded by the Offnao debugger. All the pictures below are screenshots taken from Offnao for some specific scenarios set up to show the capabilities and limitations of the algorithm.

4.2.1 Experiment 1 - Straight Lines

The first experiment involves placing the robot about a metre away from any of the 4 corners on the field, facing the corner. There are two different Offnao tabs shown below in Figure 4.4. The

Algorithm 6 RANSAC Line and Circle

$i \leftarrow 0$
$radius = 600$
$bestLineVariance \leftarrow max$
$bestCircleVariance \leftarrow max$
while $i < k$ **do**
  $p1 = points[rand\%size]$
  $p2 = points[rand\%size]$
  $line = line(p1, p2)$
  for all $p \in points$ **do**
    $distance = \frac{|line.t1 * p.x + line.t2 * p.y + line.t3|}{\sqrt{line.t1^2 + line.t2^2}}$
    if $distance < e$ **then**
      $variance \leftarrow variance + distance$
      $numPoints \leftarrow numPoints + 1$
    end if
  end for
  $variance \leftarrow variance * 0.2 - numPoints$
  if $variance < bestVariance$ **then**
    $bestVariance = variance$
  end if
  $circle = circle(p1, p2, radius)$
  for all $p \in points$ **do**
    $c \leftarrow circle.centre$
    $distance = \sqrt{(c.x - p.x)^2 + (c.y - p.y)^2} - radius$
    if $distance < e$ **then**
      $variance \leftarrow variance + distance^2$
      $numPoints \leftarrow numPoints + 1$
    end if
  end for
  $variance \leftarrow variance * 0.2 - numPoints$
  if $variance < bestVariance$ **then**
    $bestVariance = variance$
  end if
  $i \leftarrow i + 1$
end while

Algorithm 7 Find Lines and Circles Algorithm

while $RANSACLinesAndCircles(points)$ AND $points.length > N$ **do**
  if $lineVar < circleVar$ **then**
    $line \leftarrow RANSACLine$
    $points \leftarrow points - linePoints$
  else
    $circle \leftarrow RANSACCircle$
    $points \leftarrow points - circlePoints$
  end if
end while

first shows the vision tab where we can see what the robot sees. Here each of the coloured points represent points that have been assigned to a line, so the blue points belong to one line and the purple points belong to the other. The second picture shows the overview tab where the tiny blue dots represent the points from the previous image and the red lines on top of them represent the lines that have been detected. As you can see in this case both the lines have been detected very nicely.

Figure 4.4: Line detection as shown by the vision tab (left) and the overview tab (right)

4.2.2 Experiment 2 - Centre Circle Segment

The second experiment involves placing the robot near the centre circle so that it can see only a short segment of the circle. In the vision tab screenshot in FIgure 4.5, the black points represent circle points, whilst the white ones represent ones that weren't matched to the circle. The overview tab shows us that the correct circle has been detected to fit those points.

Figure 4.5: Circle detection as shown by the vision tab (left) and the overview tab (right)

4.2.3 Experiment 3 - Centre Circle and Halfway Line

The third experiment involves placing the robot about a metre from the halfway line, near where it intersects the centre circle. Here in FIgure 4.6 we can see in the vision tab and overview tab that the centre line and circle are both detected as separate features in the same frame. In the vision tab, the black points represent the circle points whilst the purple points belong to the halfway line.

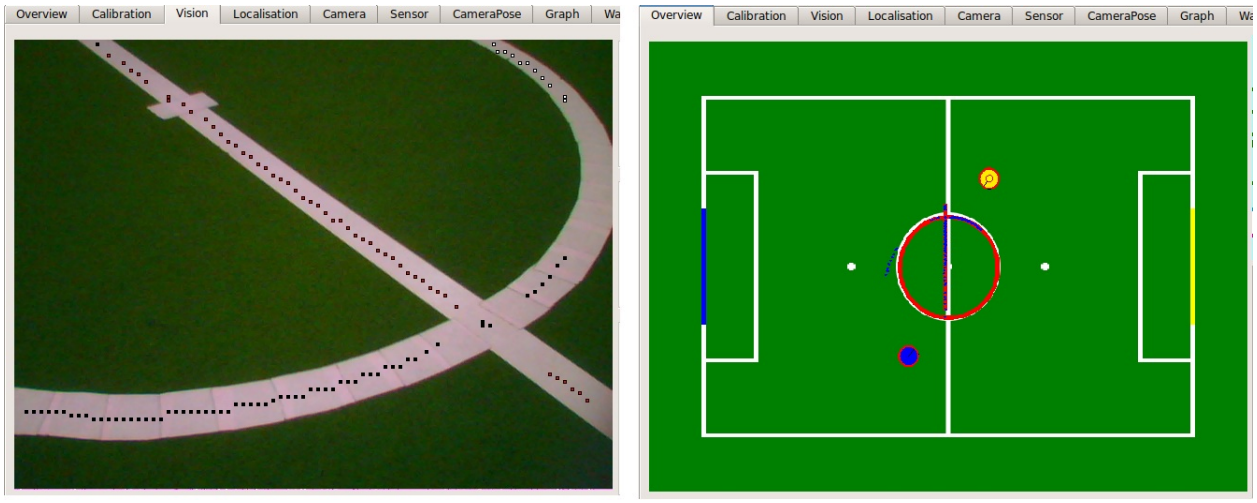

Figure 4.6: Simultaneous circle and line detection as shown by the vision tab (left) and the overview tab (right)

4.2.4 Experiment 4 - Double Line

The fourth experiment shows the limitations of this approach. It involves placing a robot near the limit at which it can detect a line, so approximately 1.5m from the goal line. As you can see from the vision tab in Figure 4.7, the points on the goal line are two different colours, indicating that they have been detected as 2 separate lines, despite actually all being part of the goal line.

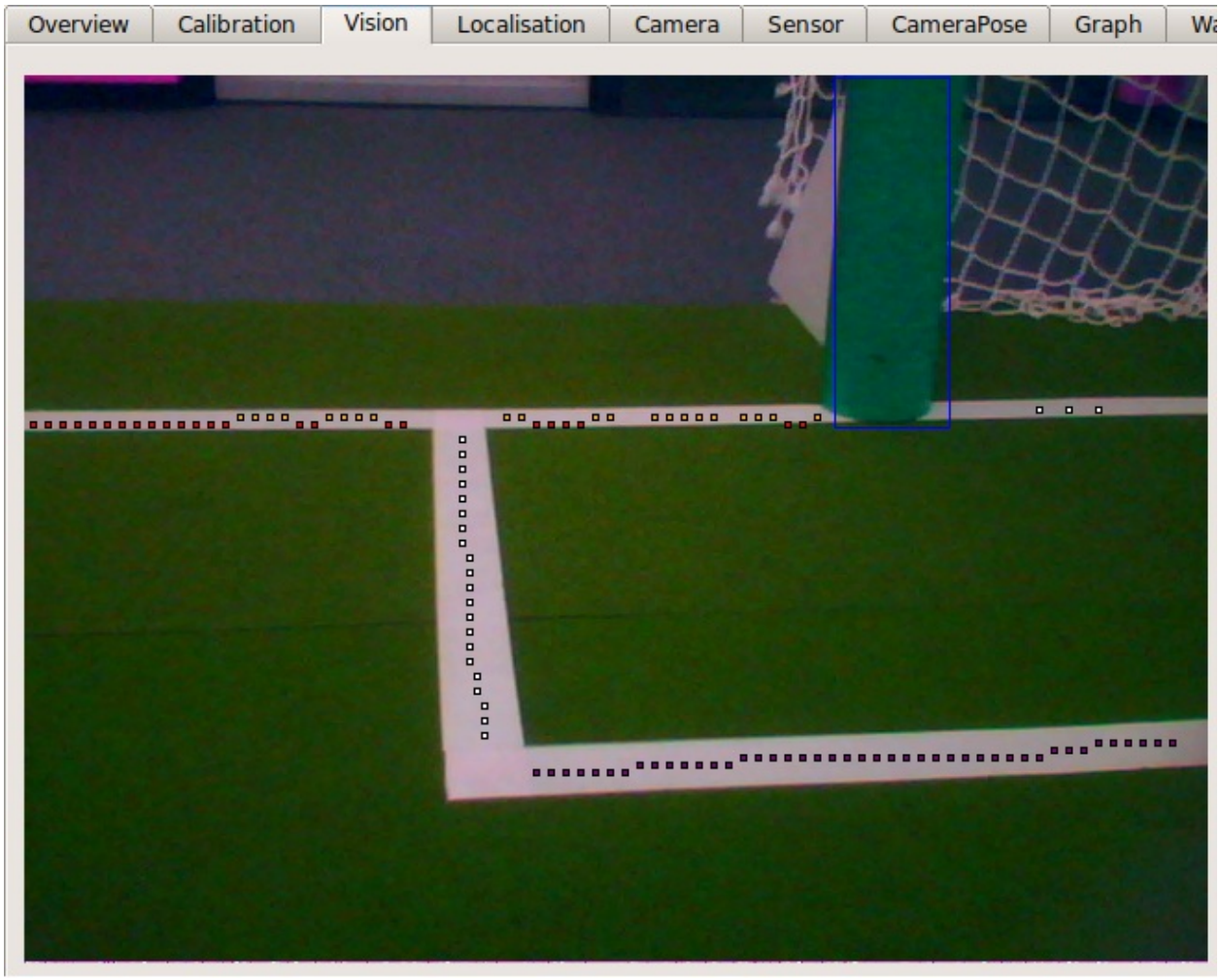

Figure 4.7: The red points belong to one line whilst the orange points belong to a separate one

4.2.5 Experiment 5 - Circle Segment Detected as Line

The fifth experiment shows the limitations of the algorithm to detect differences between very short segments. It involves placing a robot such that it can only see a very small part of the centre circle. As you can see from the screenshots in Figure 4.8, the short segment is incorrectly identified as a short line segment rather than a circle segment.

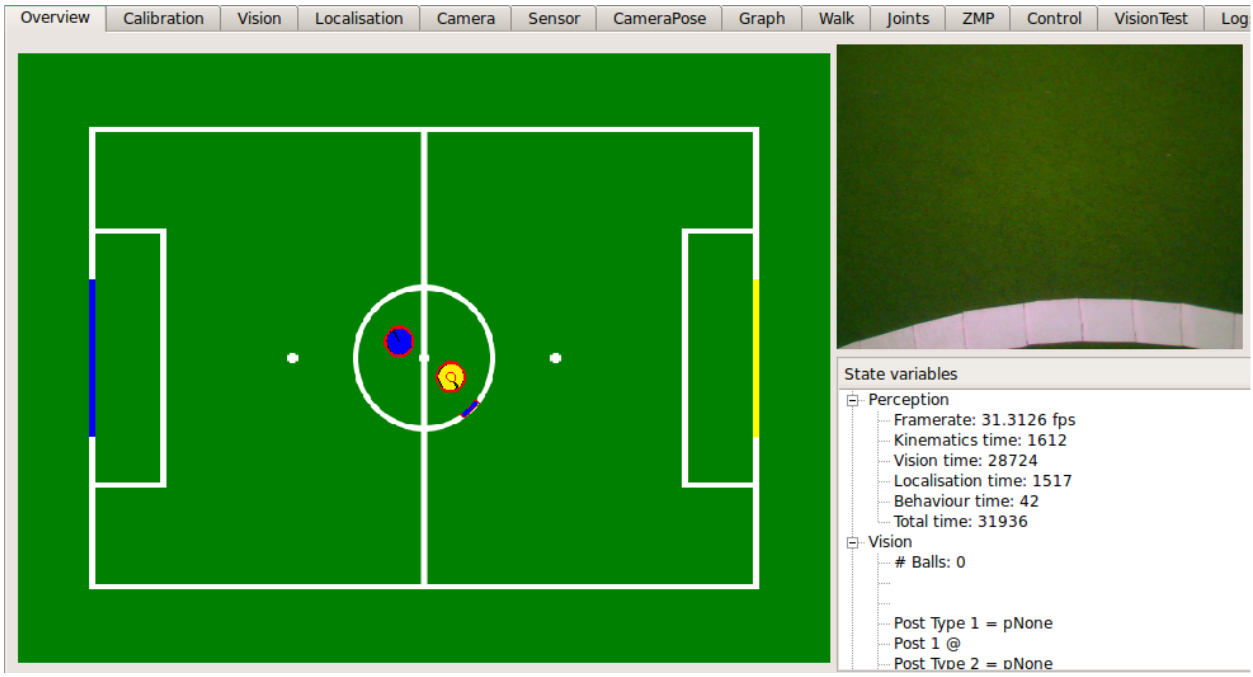

Figure 4.8: The segment is classified as a short line as opposed to a piece of the centre circle

4.2.6 Experiment 6 - Goal Box

The sixth experiment involves placing a robot between 1 and 1.5 metres from the goal line, near the corner of the goal box. At approximately 1.3m from the goal box, the shorter vertical segment doesn't get enough points to fulfil the minimum number of points and doesn't get classified as a line. This is demonstrated in Figure 4.9 where you can see there are lots of blue points in the area, but no red line over the top.

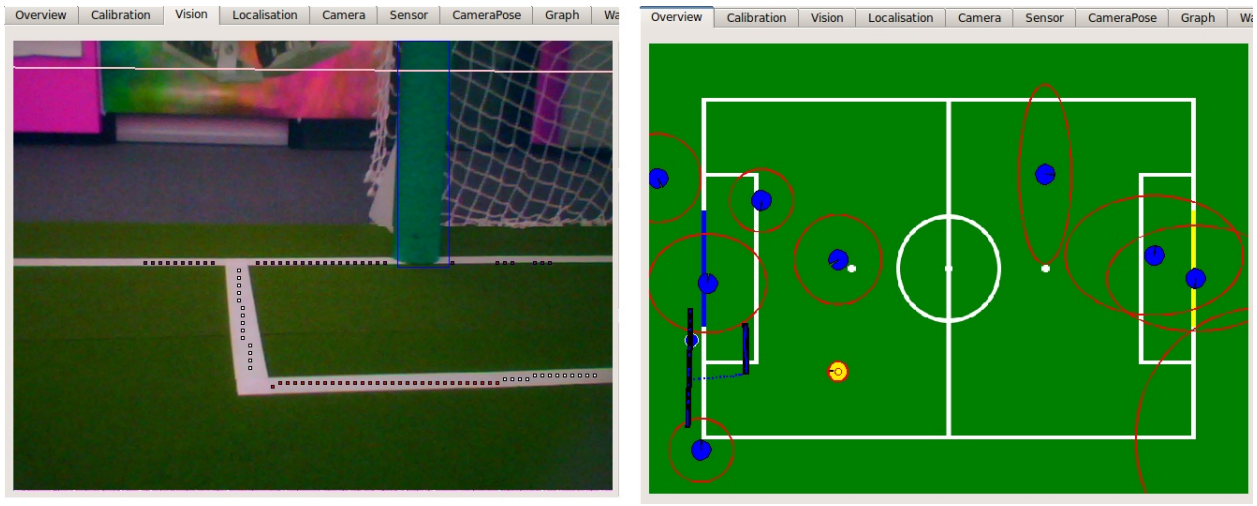

Figure 4.9: Line detection is unable to match a line to the short goal box segment

4.2.7 Metrics

There were quite a few metrics used in this section of the algorithm as well. Here is a list each of their values and an explanation of the reasoning behind each one.

RANSAC Parameters

k: Maximum number of iterations of RANSAC: 40

This value is essentially a trade-off between accuracy and speed. The more runs it gets to do, the high the probability it will pick the inliers and find a good match, but also the longer it takes. It was calculated that 40 was a good balance where the accuracy was quite high but the algorithm still ran in an acceptable time frame.

e: Maximum distance a point can be from the shape and still be considered an inlier: 15

This value is a balance between identifying points that are actually part of a shape, and not including noise or points that are part of other shapes. For this value it was determined that it would be better to go with a tighter threshold rather than a looser one, to ensure that if shapes were detected they were good quality ones and weren't skewed or affected by noise. It is also a difficult value to choose because it influences decision as to whether the shape is a line or circle. This value of 15 gave the best amount of correct identifications in combination with the rest of the RANSAC parameters.

n: Minimum number of points needed to form a line or circle: 20

This value is again a balance between not excluding short, correct lines and not matching noise like robot arms as lines or circles. Again it was better to go with a safer option of 20, which meant that noise like robot points didn't generate any lines. It was also good at not identifying short segments where it was really difficult to differentiate between lines and circles simply due to not enough information. The downside of this though was that short lines like the goal box vertical often got missed.

Variance for lines and circles: v = (sum of distances to each point / 100) - number of points

This value was determined by a lot of trial an error. A few varieties were tried, including using the sum of distances squared instead of just the sum of distances or even using just the number of points, but in the end this formula gave the most consistent measure for comparing lines and circles with themselves and with each other.

4.3 Evaluation

As you can see in the results section, the RANSAC circle and line approach worked reasonably well and was accurate in identifying both lines and circles within 1.5 metres. Due to the limitations of the candidate point selection section of the algorithm, which struggles to detect field line points

further than 1.5m away, it wasn't possible to detect these shapes at any greater distance.

Experiment 3 is a really important result for this part of the algorithm. The entire reason that the pre-grouping approach was abandoned was because it couldn't deal with this case and incorrectly classified the circle as lines. This experiment shows how generating both shapes and considering their variances against each other gives an accurate method for differentiating between lines and circles in an image.

It is worth spending time now to examine some of the cases where the algorithm didn't perform optimally and discuss some possible causes for the results. The first experiment worth looking at is number 4, the double line case. Here the low resolution of the saliency makes the situation extremely difficult. Despite being only 1 pixel higher, the points on the top line are considered a massive 50mm away from the points below once they are projected into the ground plane. With our e value set at only 15mm these two sets of points clearly aren't going to be detected as the same line, which is unfortunate. A possible solution to this problem would be to zoom in on the field line itself and analyse it in a higher resolution.

Another interesting experiment is number 5 where a short circle segment is identified as a line. In this case there isn't really enough information to accurately decide if the segment is in fact a line or a circle. The reason is is detected as a line is that the RANSAC approach has a slight bias towards choosing a line. The main reason for this is that individual lines weren't utilised by the localisation system unless they formed some sort of feature like a corner. Circles on the other hand were utilised and added weight to any hypothesis where the robot is near the centre of the field. With this in mind, the bias was added to ensure that false positive circles didn't occur, but the trade-off was that occasionally short circle segments were classified as lines instead.

Experiment 6 shows the result of another of the trade-offs in the algorithm. Here the goal box line isn't identified despite getting a somewhat healthly number of candidate points. The reason for this is the high value of the RANSAC n variable, which determines how many points are required to form a line or circle. This value was set high to ensure that noise like robot points (in particular their arms) didn't cause any false positive lines. As demonstrated in the experiment, there is a down side to that high value in that short lines can sometimes be missed simply because they don't generate enough candidate points.

Chapter 5

Detecting Field Features

This is the final stage in detecting field features. At this point the algorithm has successfully identified points that might be part of field lines, matched lines and circles through these points and now has to make as many useful landmarks out of the lines and circles as possible.

5.1 Methodology

This section of the approach is essentially an open ended task with the goal being to form as many landmarks as possible. The ones that we are able to detect are corners, T-intersections, pairs of parallel lines, the centre circle and halfway line, and of course just simple circles and lines. This section will explain each of the landmarks and the methodology to go about detecting them.

5.1.1 Intersections - Corners and T's

The first step with detecting both corners and T's is to find intersections of lines. An important concept to consider here is that lines are represented in our system as just that, lines, not segments. While the difference is subtle, segments have defined endpoints, whilst lines do not, meaning that an intersection of two lines might not accurately represent two field lines intersecting, nor the nature of their intersection. To help combat this, the list of points corresponding to each line is stored so that rough endpoints can be calculated.

The process for finding these intersections involves scanning through the list of lines detected and seeing if each one intersects any of the other lines. Whilst this approach is technically $O(n^2)$ there are so few lines detected in one frame that the time complexity isn't a factor, especially since the calculations done on each are quite fast. The method for calculating the intersection between two lines is to put them into two matrices and perform an LU decomposition, which is essentially just an optimised method for solving linear equations. If the linear equations have a solution then the lines intersect and we know the point of intersection.

Some sanity checks are applied at this point of the algorithm. For the point of intersection to be considered for further examination it has to occur not only within the image, but also at least 8 pixels from the edge. The two lines that are being intersected also need to be roughly perpendicular as well, otherwise they can't be a corner or a T-intersection.

If the sanity checks are passed, the next step is to identify what sort of intersection we are dealing with. Since lines aren't represented as segments, it is difficult be certain of exactly where they end, making it tricky to determine if an intersection is a corner or a T-intersection. The method used here to differentiate between the two relies on the idea that in a corner all the points belonging to one line should lie on one side of the other line, and vice versa. In a T intersection though, one line should have all its points on one side of the other, whilst the other line should have points on both sides of the first line. With this in mind, the algorithm for determining the if we might have a T-intersection is as follows:

Algorithm 8 Calculating Intersection Type

$points \leftarrow line2points$
for all $p \in points$ **do**
  $distance = \frac{|l1.t1 * p.x + l1.t2 * p.y + l1.t3|}{\sqrt{l1.t1^2 + l1.t2^2}}$
  if $distance < 0$ **then**
    $negativeCount \leftarrow negativeCount + 1$
  else $\{distance >= 0\}$
    $positiveCount \leftarrow positiveCount + 1$
  end if
end for
if $(positiveCount > N)$ and $(negativeCount > N)$ **then**
  return TRUE
else
  return FALSE
end if

This algorithm is run twice, one with the points from line2 against line1, then with the points from line1 against line2. Afterwards, if both times it returns false then it is probably a corner, if only one returns true then it is a T-intersection and if both return true it has detected some sort of X-intersection that must be wrong. The only other check done is to make sure that the lines intersecting are perpendicular. This is also a fairly simple calculation of the angle between two lines using atan. Once the intersection type has been confirmed and all the sanity checks have been passed, the point of intersection is transformed into the ground plane and stored.

The next step now is to calculate the distance, heading and orientation of the feature. Calculating the distance to each feature is very simple since we know the points of intersection in the ground plane, it is a simple distance between two points calculation. Calculating the heading is also simple as it is just finding the angle from the robot to a point. Orientation is a little more complicated though, it involves some basic geometry and is based on a combination of the heading and the feature's rotation in space. The formulas are as follows:

$$Distance = \sqrt{intersectionX^2 + intersectionY^2} \tag{5.1}$$

$$Heading = \arctan\left(\frac{intersectionY}{intersectionX}\right) \tag{5.2}$$

$$Orientation = \begin{cases} heading - rotation + 180 & \text{if } rotation > 0 \\ heading - rotation - 180 & \text{otherwise} \end{cases} \tag{5.3}$$

The rotation of a feature is defined as being the angle between the 'primary line' of the feature and the line from the robot to the feature. For a corner the primary line is defined as being the line that bisects the corner, whilst for a T-intersection the primary line is defined as being the upright line in a T. Figure 5.1 shows the (x, y, orientation) of four corners.

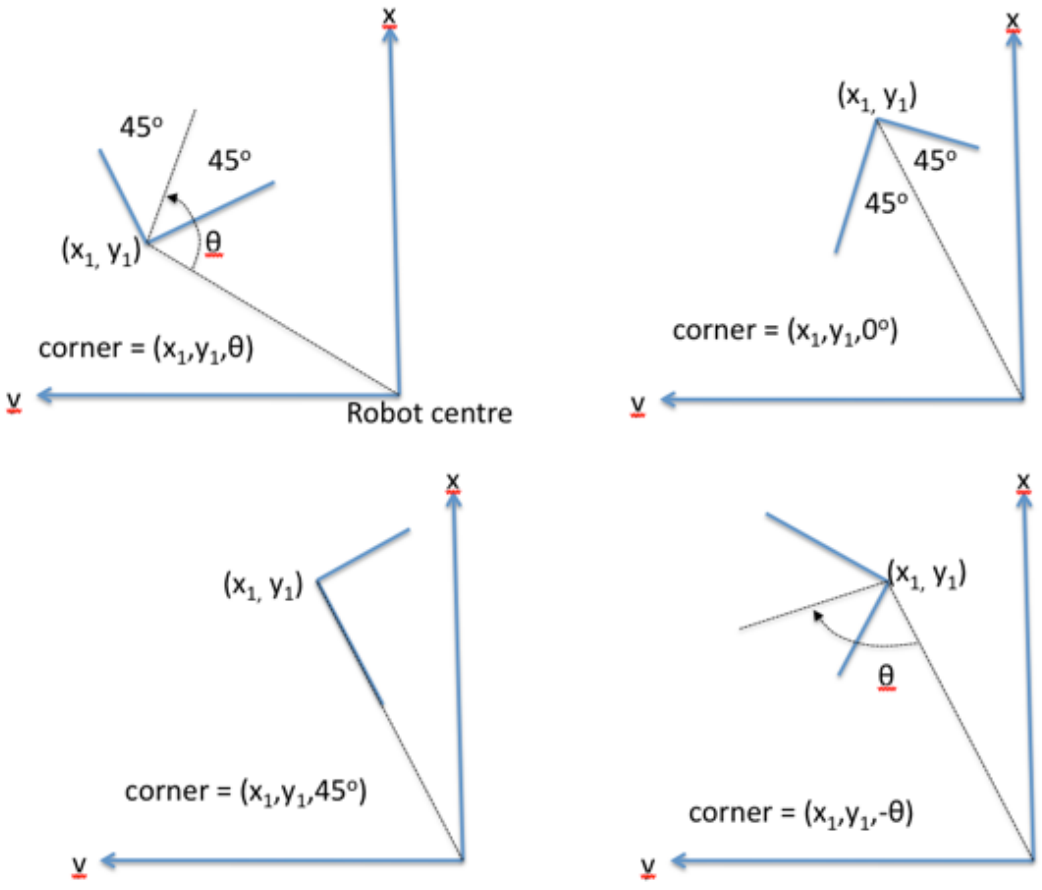

Figure 5.1: Four examples of calculating orientation for corner intersections

5.1.2 Centre Circle

Most of the work for the centre circle detection has already been done by detecting the circle in the RANSAC section. It calculates the location of the centre of the circle in the ground plane, meaning that the distance and heading can already be calculated. The only thing we can't work out with

just the circle is its orientation, which requires detecting the halfway lines as well. If we do however detect both the circle and the halfway line in one frame, the method for calculating its orientation is just a simple calculation of the angle between two lines again. We form one line between the base of the robot and the centre of the circle, and the second line is the detected halfway line. The angle between these two lines (in the range 0-pi) gives us the orientation. The exact formulas are as follows:

$$Distance = \sqrt{centreX^2 + centreY^2} \tag{5.4}$$

$$Heading = \arctan\left(\frac{centreY}{centreX}\right) \tag{5.5}$$

$$Orientation = robotToCentreDirection - centreLineDirection \tag{5.6}$$

The method for finding the direction of the two lines used for the orientation is just a simple vector direction calculation similar to the following.

$$Direction = \arctan\left(\frac{vectorY}{vectorX}\right) \tag{5.7}$$

5.1.3 Parallel Pair

If any straight lines form perpendicular intersections, they are removed from the list of lines by the time this point in the algorithm is reached. Thus the only lines passed to this function must not intersect at right angles. With this in mind, the first thing to look for is lines that are roughly parallel. This is a simple calculation where we look at the angle of the two lines relative to the robot and see if they are within a threshold value. If they are, then we also need to check how far apart they are from each other. Since this feature is representative of the goal line and the edge of the goal box, the lines should be roughly the same distance apart as those two lines are, which is 600mm. To calculate this we simply take the perpendicular distance to each line and check that they are roughly 600mm different.

If both of the above sanity checks are passed, then we can be satisfied we have a parallel pair and the information is stored to be passed onto localisation. The information that we do pass on for this feature includes the data for the closest line as well as the perpendicular distance and heading to this first line. They are calculated as follows:

$$PerpendicularDistance = \frac{|line.t3|}{\sqrt{line.t1^2 + line.t2^2}} \tag{5.8}$$

$$Heading = \arctan\left(\frac{-line.t1 * line.t3}{(line.t2 * line.t3}\right) \tag{5.9}$$

Note that there isn't an orientation for this feature, so that field is left as NaN so localisation knows not to use it.

5.2 Results

5.2.1 Experiment 1 - Corner

The first experiment involves placing a robot looking at one of the corners about 1m away. As you can see from the overview tab in Figure 5.2 where the yellow circle is the robot, we used the one of the goal box corners. In this scenario the robot detects both the goal line and the sideline, intersects them and matches the intersection to a corner. In the field view the red lines represent the detected lines, while the black L shape represents a corner has been detected there.

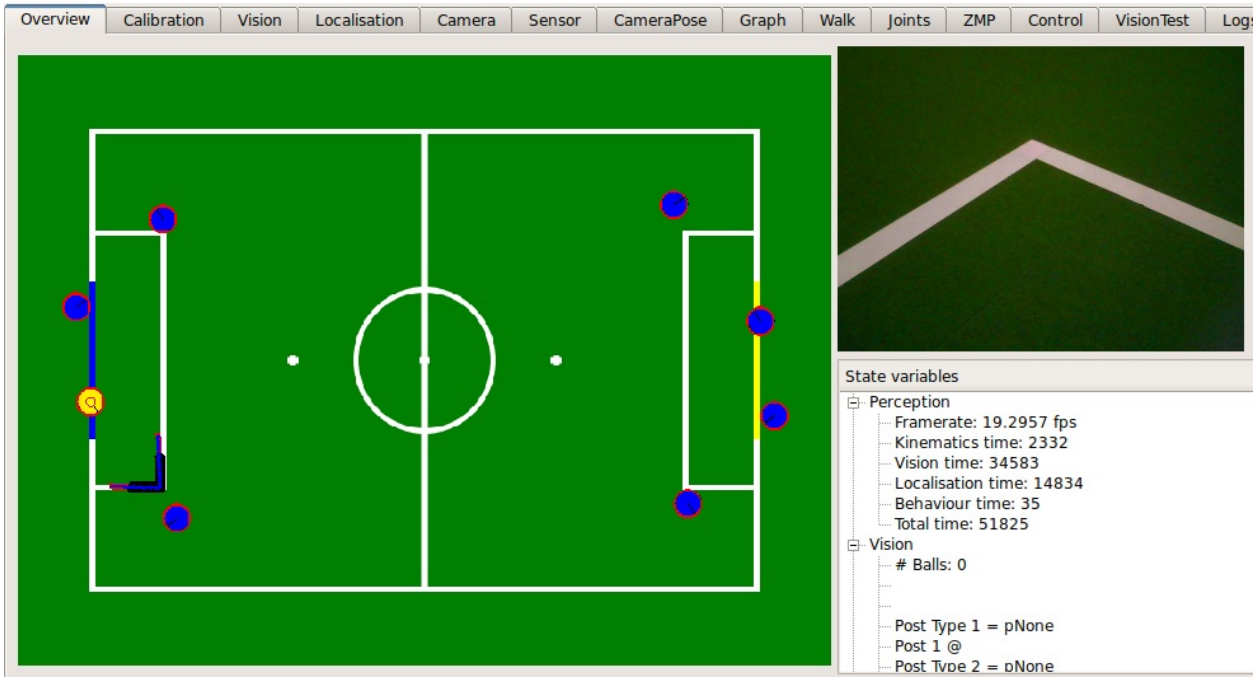

Figure 5.2: Corner detection and the corresponding modes generated by localisation around the field

5.2.2 Experiment 2 - T-Intersection

The second experiment involves placing a robot on the halfway line looking at the T-intersection formed by the halfway line meeting the sideline. As shown in Figure 5.3, both the sideline and halfway are detected and a T-intersection is formed between them. The distance, angle and heading of the observation allow the robot's position to be calculated by localisation and drawn as the yellow circle. Also notice how localisation generates a total of 6 modes, each one facing a different T-intersection around the field.

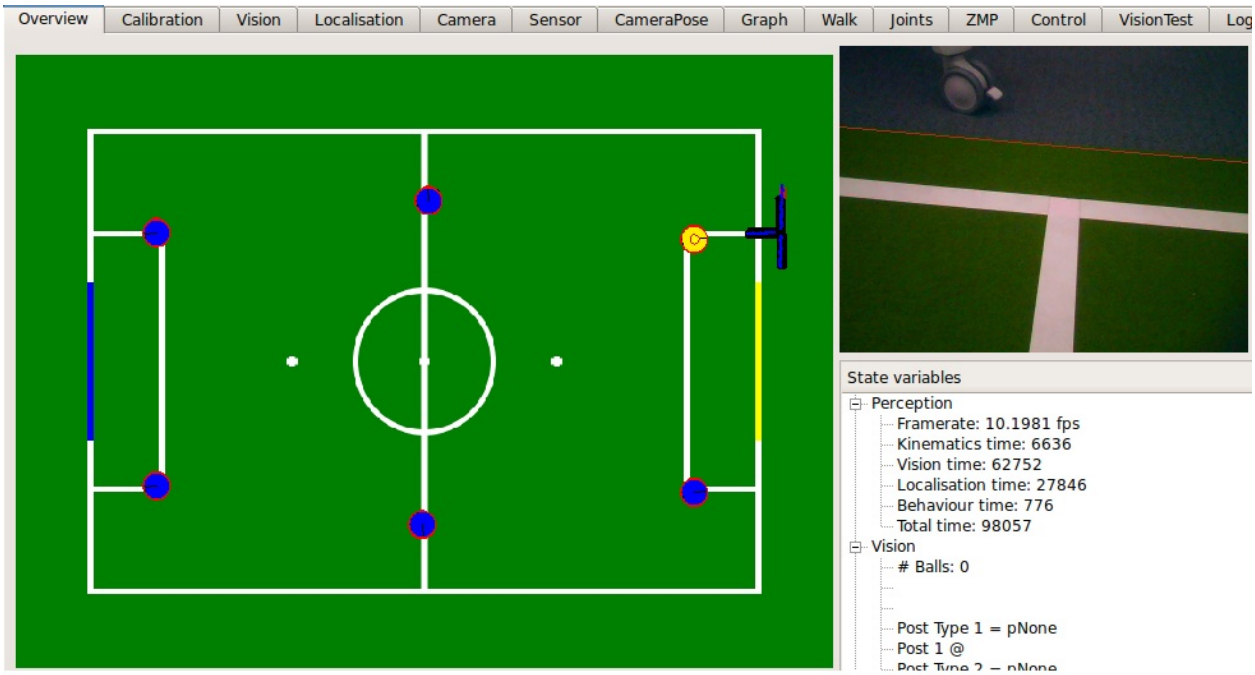

Figure 5.3: t-Intersection detection and the corresponding modes generated by localisation around the field

5.2.3 Experiment 3 - Centre Circle and Halfway Line

The third experiment involves placing a robot about a metre from the halfway line, looking towards the centre circle. From this position, the centre circle and the halfway line are both detected, as shown by the red circle and red line in the field view. Notice this time how there are only 2 modes generated by localisation, this is because this frame can only be replicated at a total of 2 positions on the field, making it an extremely useful observation.

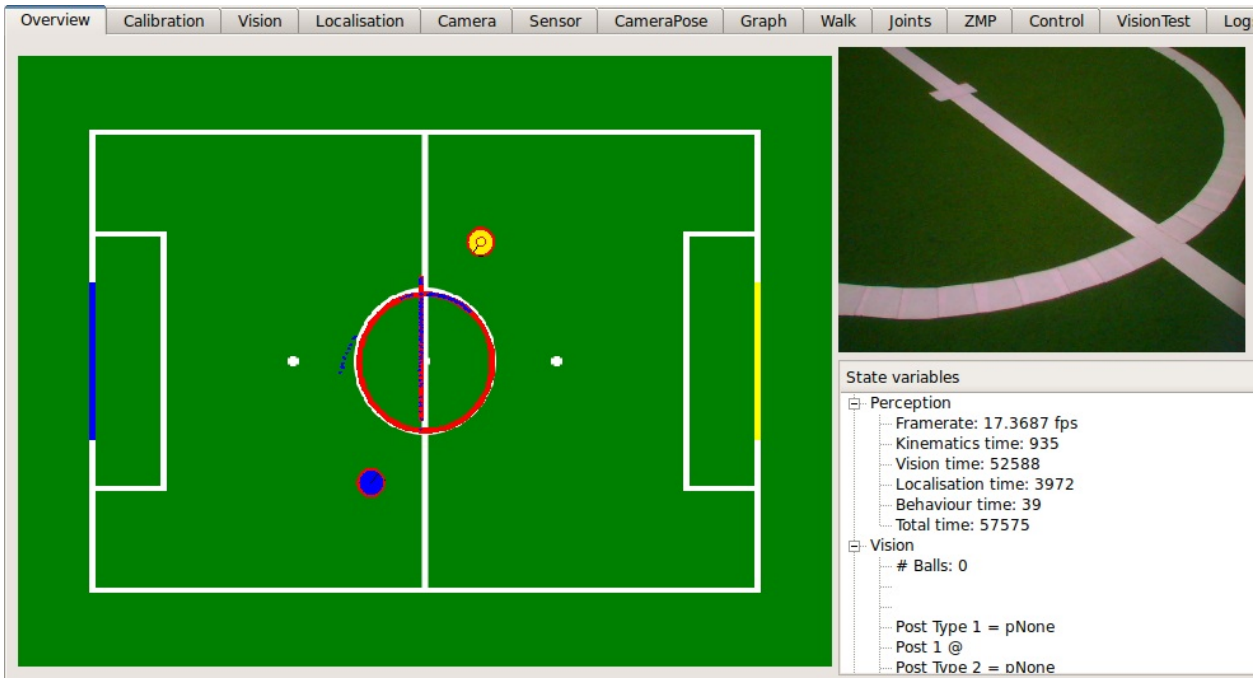

Figure 5.4: Centre circle and halfway line detection and the corresponding modes generated by localisation around the field

5.2.4 Experiment 4 - Parallel Pair

The fourth experiment involves placing a robot looking towards the goals where it can only see the goal line and the top of the goal box. These lines are detected and painted black to show that they match the criteria for the parallel pair update. Unfortunately the lines aren't enough information to generate a specific set of positions on the field, but they are useful to update the y co-ordinate and heading of the robot.

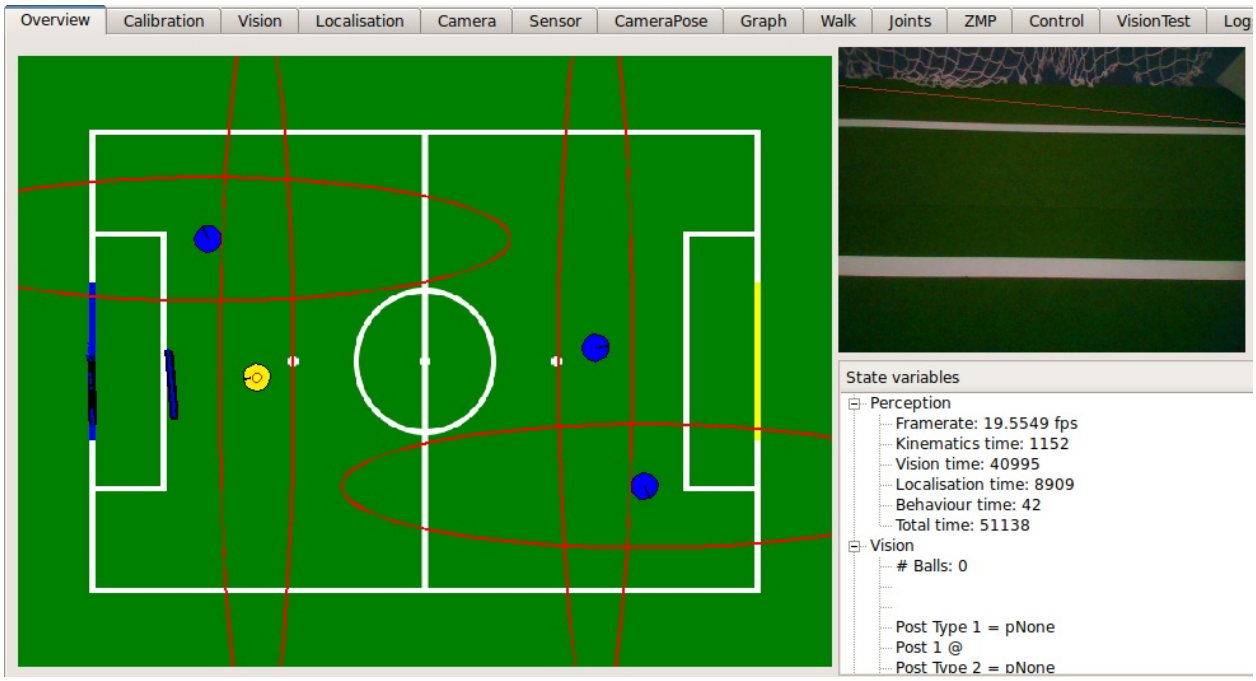

Figure 5.5: Parallel pair detection and the update applied to localisation modes around the field

5.2.5 Experiment 5 - Exclusion Zone

The fifth experiment involves placing a robot looking at a T-intersection such that the intersection occurs right at the edge of the frame. At the edge here (or even out of the frame) it is difficult to detect what type of intersection these lines form, so we go with the safe option of reporting neither. Notice on the field view that both lines are detected (in red) but no black lines have been draw to show a feature being detected.

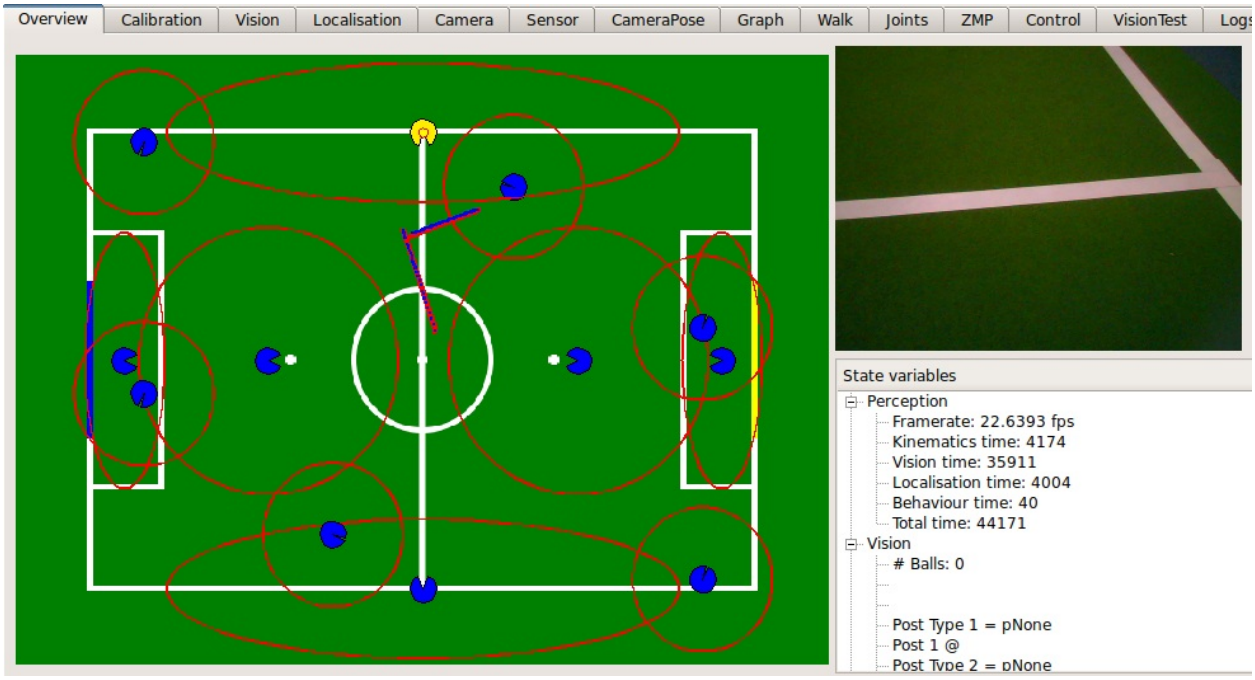

Figure 5.6: An intersection not being reported due to its proximity to the edge of the frame. Note how the robot is poorly localised due to the lack of observations

5.2.6 Metrics

Exclusion Width: 8 pixels

This metric was chosen as usual with caution in mind. Sometimes intersections that are close to the edge can be accurately determined, but often they aren't, so this was made quite high to reduce the rate of false positives. Occasionally even still there are a few incorrect classifications, but raising this value any higher would have reduced the number of correct classifications too much to justify.

Perpendicular / Parallel Lines Threshold: 7 degrees

Although 7 degrees seems like quite a tight measure, especially considering some of the candidate point selection angle thresholds, it actually turned out to be quite a lenient one. Usually lines are detected extremely accurately so this threshold isn't hit, it's only when errors in the kinematic chain calibration skew point projection that crooked lines appear.

Number of points needed on each side of a line to be considered a T-Intersection: 2

This threshold was kept very biased towards T-intersections since the candidate point selection algorithm doesn't label any green pixels as possible field line points. This means that the only time points on a corner could by on the other side of the intersecting line is right at the point of intersection. Thus there was an allowance for 1 point to be on the wrong side of the line but no need to allow any more than that. Even with the bias towards T-intersections, nearly all incorrect intersection classifications are in the form of T-intersections being classified as corners.

5.3 Evaluation

One of the unique features of this design is the parallel pair feature. It came about because of the need for more landmarks when the robot was located near the goals, but too far away to pick up the corner or T-intersection at the corner of the box. Since localisation didn't make use of just individual lines, when a pair of lines was detected all that good information was lost if the short vertical segment of the box was not detected (which was fairly often due to its short length). Thus the parallel pair was an extremely useful landmark that was developed to help counter the limitations both of this algorithm as well as the localisation system.

The improvement to the centre circle information by adding the orientation was also an unplanned development that actually got implemented at the World Championships in Turkey. It was noticed that by taking into account the halfway line it is possible to narrow the robot's position down to two symmetrical locations on the field, which actually makes it one of the most precise observations in the entire system. It is worth making a reference to David's paper here as he tallied the number of 'good' shots by the robot aiming towards the goal [4]. You will notice that after the match against NTU the number of really bad shots drops to zero for the rest of the games. This drop in the number of complete mis-localisations was no doubt related to the fact that the centre circle orientation was added immediately after the NTU game. This shows just how useful that update really is to our system.

Another important feature is the information localisation can generate if a goal post and a T-intersection are detected in the same frame. This occurs when the robot is near one of the goal posts and gets a high number of somewhat ambiguous updates like a one post update or a corner update. While these observations are good if the robot is well-localised, if it is completely lost they aren't enough to localise it again quickly. However if a goal post and the T-intersection next to it are detected in the same frame, not only can we tell which post we are looking at, we can use the orientation of the T-intersection to pinpoint the exact position of the robot on the field. This is especially useful for the goal keeper who essentially can't see two accurate goal posts at once and thus struggles to get relocalised after a dive.

Overall the final result was pretty close to what we had set out to develop, with the only planned feature that didn't get implemented being the penalty spot. The most important aspect of the field line model was to report minimal false positive features as they wreak havoc with localisation and this was most definitely achieved by always erring on the side of caution.

Chapter 6

Speed

This chapter will present the results of a variety of speed tests run on the robot in game conditions. It will also provide some discussion about the significance of these times and speculate on reasons for the results.

6.1 Results

The following graphs show the time taken by the Field Feature Detection run under game conditions. In each of these tests the ball was placed in the centre of the field and a robot was placed at various locations within its own half. The results are as follows:

6.1.1 Experiment 1

The first experiment involves placing a robot on the penalty spot, inside its own half of the field, facing the ball. The ball should be placed in the middle of the centre circle. In this experiment the robot searches for the ball, finds it and walks towards it. As it walks towards the ball it also gets the centre circle and halfway line in the frame, which corresponds to the large hump of activity in the RANSAC Lines and Circles graph in Figure 6.1. It then lines up a shot looking at the goal posts, shoots and scores. The last little hump of activity at the end shows the robot looking around for the ball again and seeing the centre circle segments near its feet.

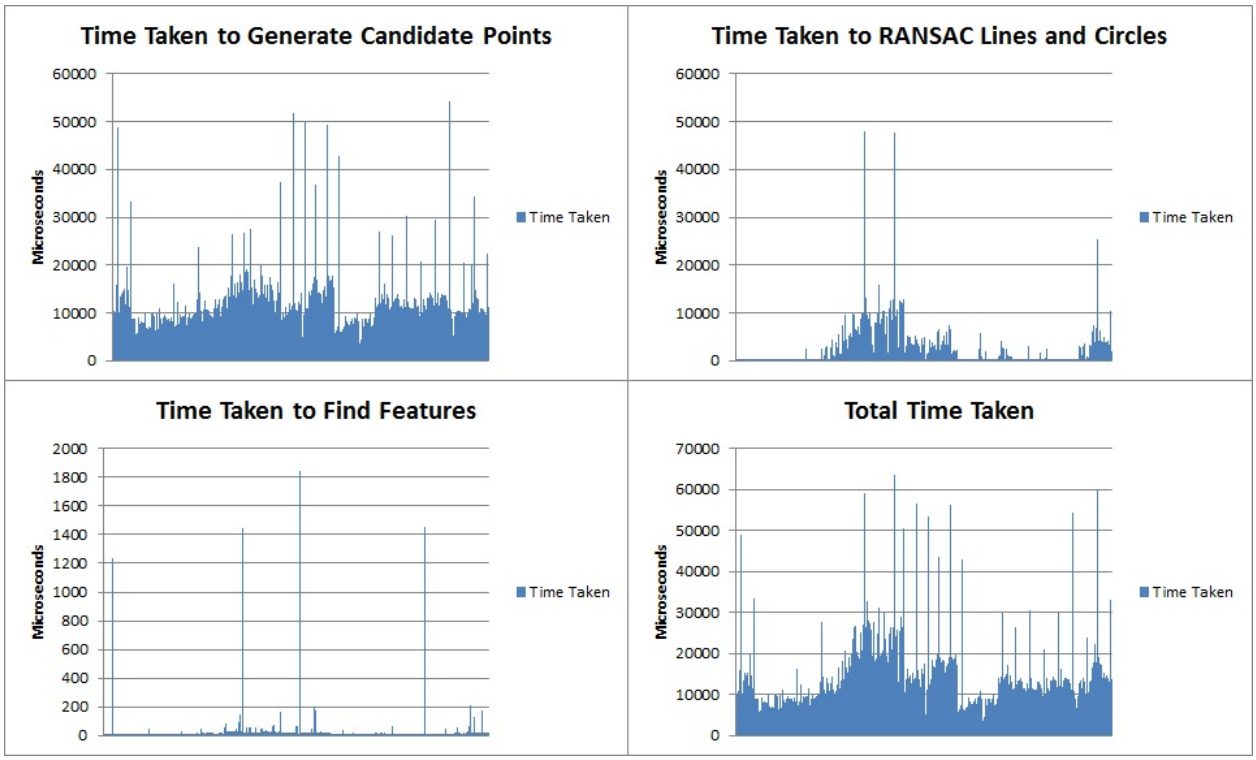

Figure 6.1: Graphs showing the time taken for each Field Line Detection module

6.1.2 Experiment 2

The second experiment involves placing a robot in line with the penalty spot, on the side line, inside its own half of the field, facing inwards. The ball should be placed in the middle of the centre circle. In this experiment the robot walks up to the ball, detecting the halfway line and centre circle on the way. This time corresponds to the first large hump in the RANSAC Lines and Circles graph in Figure 6.2 where the system is running RANSAC on the centre circle points.

Then the robot does a side kick of the ball which stops near the corner of the goal box. The somewhat empty section in the RANSAC Lines and Circles graph corresponds to this time as the robot walks across the edge of the circle and the goal box is too far away to detect. Once the robot gets close to the goal box the other big hump starts, showing the line and corner detection running. This section lasts a bit longer than the circle section did because the robot takes longer to line up the shot from here. After this the robot shoots and scores so the experiment ends.

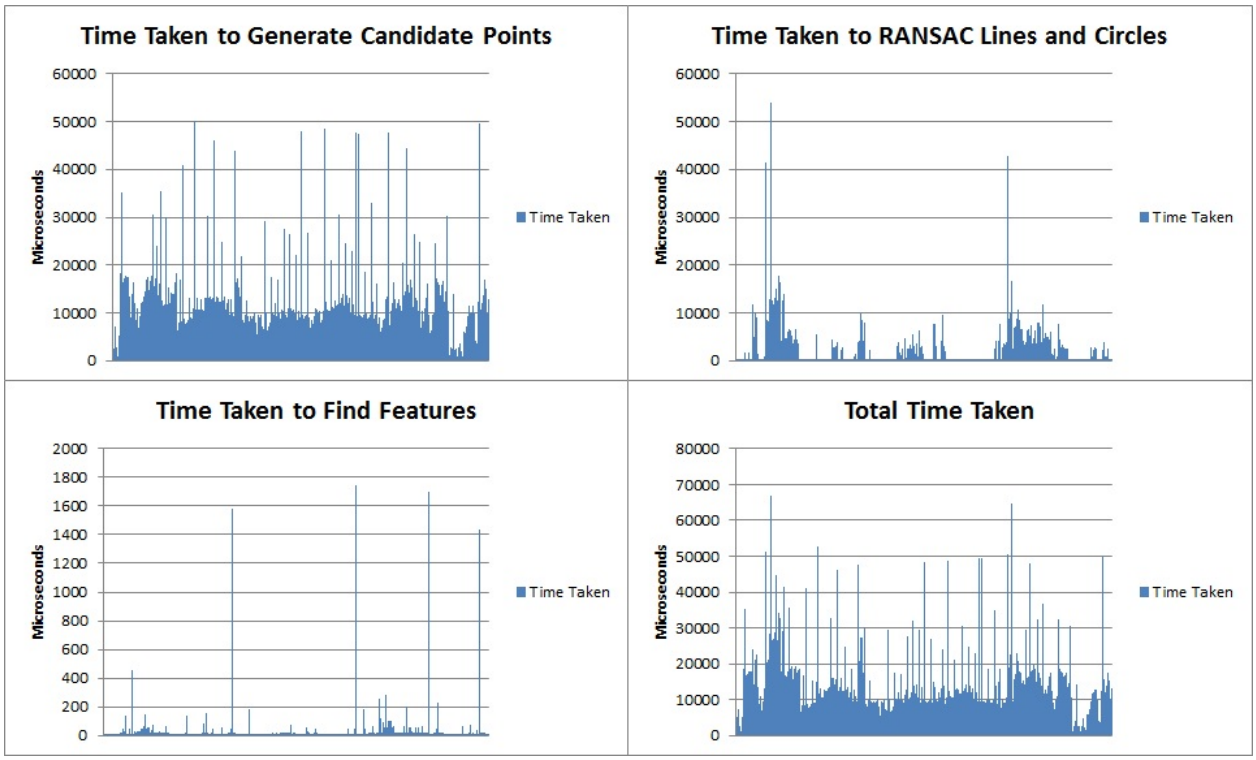

Figure 6.2: Graphs showing the time taken for each Field Line Detection module

6.1.3 Experiment 3

The third experiment involves placing a robot in the same location as experiment 1, but this time facing its own goals. Again the ball is placed in the middle of the centre circle. In the experiment the robot does a full head scan looking towards its own goals, detecting the lines and intersections associated with that region in the process. This corresponds to the first 3 sections of activity on the RANSAC Lines and Circles graphs in Figure 6.3. The robot then rotates about 120 degrees and starts another head scan, at which point it detects the ball and starts looking and walking towards the centre circle. The robot then spends a fair amount of time lining up the shot before finally shooting and scoring. This approach and line up phase is reflected by the long section of activity in the middle of the graph. The final section of activity is the robot restarting its find ball scan and seeing the centre circle segments near its feet.

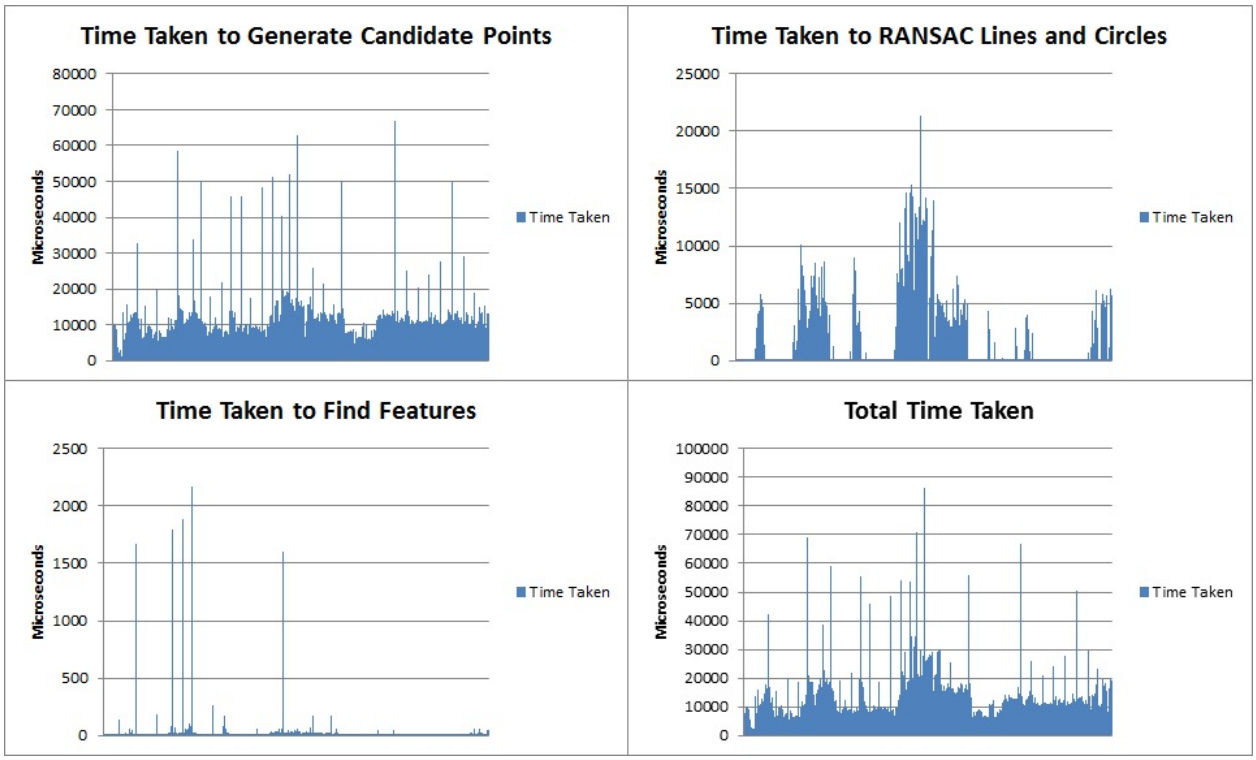

Figure 6.3: Graphs showing the time taken for each Field Line Detection module

6.2 Discussion

One thing to note about these times is that they aren't 100% accurate. They are measured using the timer class, so if a thread gets pre-empted then the timer continues counting while the other thread is running. A clear example of this is in the Time Taken to Find Features graphs where it usually runs in far less than 100 microseconds, yet occasionally it spikes up and runs for up to 2 milliseconds. In this case the thread manager pre-empts the vision thread and instead runs motion whilst the timer still counts.

One of the interesting observations to note here is that the candidate point selection algorithm takes a minimum of about 5 milliseconds and a rough average of 10 milliseconds. From looking at the graphs, you can see that when there aren't any features in the frame, this is the only section that actually runs and in these cases it usually takes less than 10 milliseconds. This is important because it frees up lots of time for the other detectors, which are more likely to have work to do if the there aren't any field lines in the frame. For example, frames with large sections of goal post in them don't usually have a large volume of field line features in them.

Another positive result is that the RANSAC Lines and Circles function rarely takes more than 10 milliseconds to run, even when there are frames with lots of lines and circles to detect. The average time though is a little higher than we would have liked. The graphs show the average time to be between 5-10 milliseconds, depending on how many features are in the frame, which when

combined with the candidate point selection algorithm means the entire processes takes around 20-25 milliseconds on busy frames. This unfortunately doesn't leave enough time to run all the other detectors and still completed the perception cycle in less than 30 milliseconds, meaning that frames are lost. However the accuracy of the field feature detection somewhat makes up for the missed frames by rarely not detecting features in the frame.

One extremely positive result is the speed of the Find Features section. It runs with an average time of around 50 microseconds, which is extremely fast, especially considering the speed of the previous two sections. This means that even when there are lots of intersections and features to construct in a frame, this step of the algorithm doesn't add any more time overheads.

Another interesting result is the occasional huge peak in time taken. The boost in time in the Find Features graph is easily explained as motion pre-empting the thread for around 2 milliseconds, but the others have much larger peaks that can't be attributed to motion so easily. To be honest we aren't entirely sure why there are such huge peaks in the times overall, but we do have some suspicions. It could be that the field edge detector fails to find an edge and the entire image is scanned and processed. If the image was actually of the crowd next to the field this frame could have a huge number of edges and might explain the large peaks in time. Another idea is that it might be the networking or infrastructure threads having a sudden burst of work to do, but again there is no direct evidence that this occurs.

Overall the speed of the algorithm didn't quite reach the benchmarks set, which was unfortunate. Due to the inability to run a profiling program on the robot, it was nearly impossible to determine exactly which parts of the code were causing the slowdown. This in turn made it very difficult to fully optimise the code as we weren't able to pinpoint the areas to focus in on. In saying that, the accuracy of the features detected meant that when this section runs in full there is a lot of good quality information generated, so missing some of the frames wasn't nearly as detrimental as first thought.

Chapter 7

Future Work

There are a variety of ways this development could be extended including adding more features, increasing the distance we can detect features as well as making the system more robust to noise. This chapter looks at some specific avenues of work that would give the most improvement to the current system.

7.0.1 Detecting the Penalty Spot

This was the only major field feature not implemented primarily because it didnt fit in the with lines and circles model. The candidate points detected are far too few to ever get classified as a line or a circle, so they are always just thrown away after the RANSAC stage.

Any sort of penalty spot detected would be difficult to implement at a distance of any more than a metre or so, simply due to the low number of candidate points it would generate. It would be quite difficult to differentiate the penalty spot from a bit of noise or even a robot section with so few points.

At a closer distance however, the penalty spot generates a fair number of points which actually resemble a cross shape. These points could be grouped together using some sort of simple clustering algorithm or blog detection algorithm but it might take some worth to not generate false positives inside robots. There might even be enough information at close range to implement a cross based version of RANSAC to match the shape of the points to a cross.

7.0.2 Utilising Foveas

Another potential avenue of improvement would be to utilise the foveas developed by Chatfield [3] to detect far away balls to also detect far away field lines. These foveas allow you to zoom in on a particular section of each image for high resolution analysis and would help remove a lot of the errors generated by the low resolution of the saliency. This would allow field lines to be detected at far greater distances and could even improve the accuracy of closer field features detected.

This would also come with its own set of challenges though, as some sort of algorithm would need to be developed to decide which area of the image is worth investigating at higher resolution. The way the vision system is currently set up, if a good algorithm for picking the area of the original image to focus on was developed, the current field line detection code could be run using the high resolution fovea instead of the low resolution image with minimal changes.

7.0.3 Unidentified Intersections

One area of huge waste in the current system is that we dont report an intersection between perpendicular lines unless it is clearly in the frame and we can identify exactly what sort of intersection it is. While this does still occur fairly often, there are also a large number of times when the point of intersection is just outside the frame, or is close to the edge so it is hard to detect precisely what type of intersection it is. Rather than just throw this information away, it could be instead labelled as an unidentified intersection and still passed on to localisation.

This change could be implemented now with only a few lines of code in field line detection to instead save intersections sanity checked out because they dont land in the middle of the image. The problem with adding this is that the localisation module would require a fair bit of work to also accommodate the changes.

7.0.4 Utilising Edge Information

The algorithm presented earlier in section 4.1.1 that bucketed candidate field line points based on their edge directions looked extremely promising except for the circle case. If the problems with attempting to detect circles could be fixed, then the bucketing could be utilised for straight lines where it worked so well. Using the bins not only improves the accuracy by weeding out the noisy points but it also does most of the work for the RANSAC algorithm which then has to essentially just match a line to a set a points with few to no outliers. This would allow for huge speed ups in the the simple shape detection section.

Chapter 8

Conclusion

This thesis has presented a creative and efficient methodology for detecting field line features using RANSAC. At each stage of the process we have been able to achieve accurate and reasonably fast results which, when combined together, form a pipeline to detect a variety of field line features across a soccer field.

Chapter 2: Background looked at the general rUNSWift code base set up as well as some of the important aspects of the vision pipeline and environment relevant to this topic. We discussed some other approaches to detecting field line features by various SPL teams and the pros and cons of each approach. Finally, we looked at applications of RANSAC in other sections of computer vision and examined the lack of any attempt to apply the RANSAC approach to detecting field line features.

Chapter 3: Generating Candidate Points explained the process for generating candidate points using both vertical and occasional horizontal scans through the edge saliency image. We also examine the results of this algorithm, showing its effectiveness to generate points on lines within 1.5m and its short comings with robots in the image.

Chapter 4: Detecting Simple Shapes: Lines and Circles discussed the original attempt to pre-group candidate points based on their edge values before running RANSAC. We then looked at its successes and showed the fatal flaw in the algorithm that required it to be replaced. Finally we examined the successful approach using simultaneous RANSAC lines and circles and the achievements as well as shortcomings of this approach.

Chapter 5: Finding Field Features examined the methodology for combining simple lines and circles into recognisable landmarks from a soccer field. We discussed the various situations and features we can build as well as their usefulness and accuracy for localising a robot.

Chapter 6: Speed looked at the speed of the algorithms from Chapters 3,4 and 5. We examined a variety of situations to show both the minimum and maximum speeds this process can run at and discussed factors that influence the speed of the algorithm.

Chapter 7: Future Work looked at a variety of possibilities to extend the current system in both its functionality and efficiency. In addition to this it also discussed some briefly explored avenues of research that looked promising but weren't fully developed due to various constraints.

Although we fell slightly short of the desired efficiency goals, we were able to develop an extremely robust, modular and accurate system for detecting field line features on a soccer field.

Bibliography

[1] Samuel Barrett, Katie Genter, Matthew Hausknecht, Todd Hester, Piyush Khandelwal, Juhyun Lee, Michael Quinlan, Aibo Tian, and Peter Stone. Austin villa 2010 standard platform team report, 2010. Available online: `http://www.cs.utexas.edu/~pstone/Papers/bib2html-links/UTAITR1101-spl10.pdf`.

[2] Jordan Brindza, Levi Cai, Ashleigh Thomas, Ross Boczar, Alyin Caliskan, Alexandra Lee, Anirudha Majumdar, Roman Shor, Barry Scharfman, and Dan Lee. Upennalizers robocup standard platform league team report 2010, 2010. Available online: `http://www.seas.upenn.edu/~robocup/files/upennalizers_team_research_report_2010.pdf`.

[3] Carl Chatfield. rUNSWift 2011 Vision System: A Foveated Vision System for Robotic Soccer. Honours thesis, The University of New South Wales, 2011.

[4] David Gregory Claridge. Multi-Hypothesis Localisation for the Nao Humanoid Robot in RoboCup SPL. Honours thesis, The University of New South Wales, 2011.

[5] Stefan Czarnetzki, Sören Kerner, Oliver Urbann, Matthias Hofmann, Sven Stumm, and Ingmar Schwarz. Nao devils dortmund team report 2010, 2010. Available online: `http://www.irf.tu-dortmund.de/nao-devils/download/2010/TeamReport-2010-NaoDevilsDortmund.pdf`.

[6] R. O. Duda and P. E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Commun. ACM*, 15:11–15, January 1972.

[7] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24:381–395, June 1981.

[8] Laurie J. Heyer, Semyon Kruglyak, and Shibu Yooseph. Exploring Expression Data: Identification and Analysis of Coexpressed Genes. *Genome Research*, 9(11):1106–1115, November 1999.

[9] Subramanian Ramamoorthy, Aris Valtazanos, Efstathios Vafeias, Christopher Towell, Majd Hawasly, Ioannis Havoutis, Thomas McGuire, Seyed Behzad Tabibian, Sethu Vijayakumar, and Taku Komura. Team edinferno description paper for robocup 2011 spl, 2010. Available online: `http://www.ipab.inf.ed.ac.uk/robocup/pubs/EdinfernoTDDoc.pdf`.

[10] Adrian Ratter, Bernhard Hengst, Brad Hall, Brock White, Benjamin Vance, Claude Sammut, David Claridge, Hung Nguyen, Jayen Ashar, Maurice Pagnucco, Stuart Robinson, and Yanjin Zhu. rUNSWift Team Report 2010 Robocup Standard Platform League. Only available online: http://www.cse.unsw.edu.au/~robocup/2010site/reports/report2010.pdf, 2010.

[11] Thomas Röfer, Tim Laue, Judith Müller, Armin Burchardt, Erik Damrose, Alexander Fabisch, Fynn Feldpausch, Katharina Gillmann, Colin Graf, Thijs Jeffry de Haas, Alexander Härtl, Daniel Honsel, Philipp Kastner, Tobias Kastner, Benjamin Markowsky, Michael Mester, Jonas Peter, Ole Jan Lars Riemann, Martin Ring, Wiebke Sauerland, André Schreck, Ingo Sieverdingbeck, Felix Wenk, and Jan-Hendrik Worch. B-human team report and code release 2010, 2010. Only available online: http://www.b-human.de/file_download/33/bhuman10_coderelease.pdf.

[12] Rico Tilgner, Thomas Reinhardt, Daniel Borkmann, Stefan Seering, Tobias Kalbitz, Robert Fritzsche, Katja Zeiber, Christoph Vitz, Sandra Unger, Manuel Bellersen, Hannah Muller, and Samuel Eckermann. Nao-team htwk team description paper, 2010. Available online: http://robocup.imn.htwk-leipzig.de/documents/tdp_htwk_2011.pdf?lang=en.