Technische Universität Hamburg-Harburg
Vision Systems

Prof. Dr.-Ing. R.-R. Grigat

Center circle detection on the NAO robotic platform for the RoboCup Standard Platform League

Project Thesis

Nicolas Riebesel

May 29, 2018

Contents

List of Figures

Chapter 1

Introduction

This thesis is written as a research project within the robot soccer team Hamburg Ultra Legendary Kickers (HULKs) for the RoboCup Standard Platform League (SPL).

To be able to play soccer the robots need to localize themselves on the soccer field. To get features for the localization algorithms the robot uses computer vision to extract field features from camera images. One important field feature is the center circle. The center circle will appear as an ellipse in the camera images. Thus this project thesis will try to use and develop computer vision algorithms designed to extract ellipses from images to use them in the RoboCup SPL competition to get better and more accurate localization features.

The next sections contain a short introduction to the NAO robot platform, the RoboCup and an explanation of the motivation and goals of this thesis. Additionally there is a short overview of existing approaches to center circle detection.

1.1 The NAO Robot

The NAO robot is developed by the French company *Aldebaran Robotics* which was bought by the Japanese *Softbank Robotics Corp.*. It is a humanoid robot of 57.3 cm hight and weighing 5.2 kg. It is equipped with a 1.6 GHz Intel Atom CPU from around 2008 and 2 GiB of RAM. The robot is currently available in the fifth version. The sixth version is announced for April 2018 with a newer CPU, more RAM and an integrated GPU.

The robot has two cameras to see its environment. One camera sits inside the forehead and one at the position of it's mouth see figure 1.1. Due to a lack of overlapping view-ports and processing power if is not possible to use stereo-vision methods.

The cameras take images with a frequency of 30 Hz each. If every frame should be evaluated the maximum processing time is 16.6 ms per frame. But due to other processes also using CPU time a maximum time of 10 ms is more reasonable.

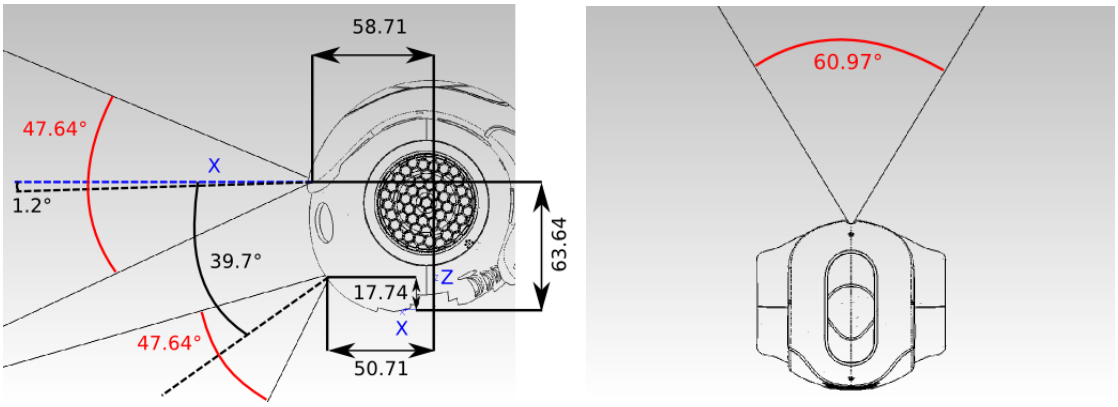


Figure 1.1: *Position and field of view of the used NAO Robot [Sof18]*

1.2 The RoboCup

The RoboCup was started 1995 as a replacement for the chess challenge which was solved in 1996 [Kit+97]. Researchers were searching for a new computational challenge and started with simple soccer simulations. Multiple teams could enter the challenge and develop algorithms to win against the other teams.

Meanwhile the scope of the RoboCup was greatly extended and also includes challenges other than soccer such as RoboCup rescue or the logistic league. Also there are now multiple soccer leagues with slightly different rules and goals. The HULKs are participating in the SPL. All SPL teams are using the same robot platform the Nao robot, see section 1.1 for more details. So the difference between all teams are the different software implementations due to com. The challenges get more difficult each year, so every team has to keep working to be able to participate in the league.

1.3 Motivation

It is an important capability for the robot to be able to localize itself on the field. The only features that are currently used for localization of a robot are the field-lines, gyroscope, accelerometer, odometry and the starting position of the robot. But the field-lines are no accurate features for localization. From the lines alone there are commonly multiple possible positions on the field that would give the same field-line measurements. If the robot would be able to detect the center circle and the center line there are exactly two positions on the field that share the same measurement. So being able to detect the center circle would greatly help to improve the localization.

In the process of the camera pinhole projection the center circle is transformed into an ellipsoid shape. Because the dimensions of the real center circle is known it is possible

to calculate the inverse transformation from a found ellipse to the corresponding center circle.

This can be used for the currently tedious camera calibration process. Finding the ellipse while playing could also enable online calibration.

1.4 Overview of existing approaches

Currently most teams rely on an image segmentation step with scanlines. The image is scanned horizontally and/or vertically to find edges in the image. Most of the time the scanlines are distributed equally along the horizontal/vertical axis and not every image line is scanned. This way it is much easier keep the computational complexity low to meet the realtime demands of the problem.

After the segmentation step many teams classify the segments in one way or another. Those segments which are identified as potential line segments are used to fit the field features into it.

The world champions of 2017 are using the current position of the joints, Inertial Measurement Unit (IMU) and the relative position of the camera to the head joint to project points of the image onto the ground [Röf+17, Section 4.3.2]. After this step the points are transformed to a top-down view of the field. Afterwards multiple lines are fitted into the data points. Because the center circle can be represented as a polygon with multiple lines a line detection algorithm normally searches for multiple lines in the points belonging to it. To find the lines belonging to the center circle, hypothetical midpoints of a center circle are calculated for every line. Afterwards all possible midpoints are clustered. If there are many points in the same region all lines belonging to it are used as the center circle.

Another team also uses a scanline based segmentation of the field. First the image is vertically and horizontally scanned for edges. Afterwards it is determined whether the color classes of the segments belong to the field lines. All line points are fitted for straight lines. Afterwards all lines are processed with a ransac ellipse fitting method that uses ellifit (see section 2.1.5) for the ellipse fitting part [Nao17].

1.5 Goals

The aim of this project thesis is to find the center circle of the SPL soccer field reliably in an image using a ellipse fitting method. The method should be able to run in less than 15 ms.

1.6 Problem Overview and Thesis Structure

The chapter 2 prerequisites describes the basic algorithms and methods used in this thesis and the used software. The chapter 3 implementation explains the details of the implementation, faced problems in the implementation and made optimizations.

The chapter 4 evaluation and discussion discusses the used dataset and validation methods. It also provides a discussion about problems of the chosen algorithm and evaluation of the runtime performance.

The last chapter 5 conclusion and outlook summarizes the thesis and gives an outlook to possible future work.

Chapter 2

Prerequisites

This chapter provides the basics needed to understand this thesis. The first section describes the mathematical primitives and the second section introduces the used software for this thesis.

2.1 Basic primitives

2.1.1 Pinhole camera projection

The pinhole camera projection is used to model the relationship between an image and the real world. It is used to describe the transformation from 3d real world coordinates to image coordinates and the reverse transformation.

Without stereo-vision it is generally not possible to convert a pixel into 3d real world coordinates. But with the assumption that everything on the image is placed on the ground it is possible to calculate the position of a pixel on the field.

For a camera image it is not possible to know the distance of the 3D point that accounts for the pixel color. The 3D point could be located on a line through the focal point and the pixel of the image sensor. If the location of the ground plane is known the intersection of this line with the ground plane can be computed.

To be able to calculate the projections the intrinsic and extrinsic camera parameters need to be known. The extrinsic camera parameters describes in which position and rotation the origin of the camera is located within a given coordinate system. The intrinsic camera parameters are dependent on the specifications of the given camera system.

In the following the extrinsic camera parameters will be contained in the rotation matrix R and the position vector P. The intrinsic camera parameters are represented by the cc (camera center) and fc (focal center) variables and are scaled with the image size.

Forward projection Forward projection is used to answer the question where a real world object would be if it can be seen in a specific image pixel.

To calculate the position of a object located in pixel $p_{\{x,y\}}$ the following equation is used:

$$cp = R \begin{pmatrix} 1 \\ \frac{cc_x - p_x}{fc_x} \\ \frac{cc_y - p_y}{fc_y} \end{pmatrix} \tag{2.1}$$

$$rp = \begin{pmatrix} P_x - P_z \frac{cp_x}{cp_z} \\ P_y - P_z \frac{cp_y}{cp_z} \end{pmatrix} \tag{2.2}$$

Afterwards the object location is at: $(rp_x, rp_y, 0)^T$.

Reverse projection Reverse projection does the reverse of the forward projection. Given a real world object, which pixels would it cover on the image?

Given an object at the 3d coordinates $rp_{\{x,y,z\}}$ the following equation yields the corresponding image coordinates:

$$cp = R \begin{pmatrix} rp_x \\ rp_y \\ 0 \end{pmatrix} + P \tag{2.3}$$

$$p = \begin{pmatrix} cc_x - fc_x * \frac{cp_y}{cp_x} \\ cc_y - fc_y * \frac{cp_z}{cp_x} \end{pmatrix} \tag{2.4}$$

The 3d point would be visible at the $p_{\{x,y\}}$ pixel.

Horizon With the help of the forward projection it is possible to calculate the position of the horizon within the image. The horizon is where all parallel lines in the perspective transformation meet. To calculate the parameters of the horizon line $(h_A, h_B)^T$ use:

$$y = h_A * x + h_B \tag{2.5}$$

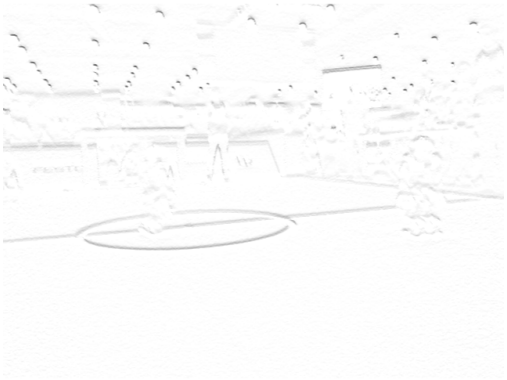
$$h_A = -fc_y R_{2,1} / (fc_x R_{2,2}) \tag{2.6}$$

$$h_B = cc_y + fc_y (R_{2,0} + cc_x R_{2,1} / fc_x) / R_{2,2} \tag{2.7}$$

with R as the rotation matrix of the camera to ground matrix.

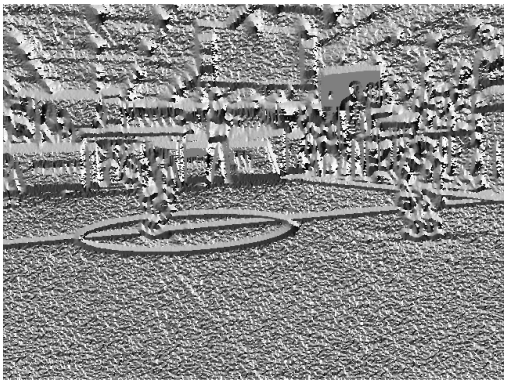
(a) *The Y channel of the original YCbCr image.*

(b) *This is the derivative in x direction of the image 2.1a.*



(c) *This is the derivative in y direction of the image 2.1a.*

(d) *This image shows the gradient of the Y channel.*



(e) *This image shows the phase of the gradient mapped to grayscale values.*

Figure 2.1: *Sobel x and y kernel applied to an image and the calculated gradient and phase images.*

2.1.2 Sobel operator

The sobel operator is a discrete operator used to calculate the derivative of an image in the x or y direction. To calculate the derivative an image needs to be convolved with either the x or y sobel kernel.

The kernel used for calculating the derivative in x direction:

$$M_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \tag{2.8}$$

The kernel used for the y direction:

$$M_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \tag{2.9}$$

To calculate the sobel derivative in either x or y direction the image needs to be convolved with the corresponding kernel. Resulting in either G_x or G_y:

$$G_x = M_x * I \tag{2.10}$$
$$G_y = M_y * I \tag{2.11}$$

Afterwards the gradient of an image can be calculated with a combination of each pixel from G_x and G_y:

$$G = \sqrt{G_x^2 + G_y^2} \tag{2.12}$$

For the calculation of the gradient phase the following equation can be used:

$$P = \arctan(\frac{G_y}{G_x}) \tag{2.13}$$

The effect of these operations on a real image is shown in figure 2.1.

2.1.3 Convex hull

Given a set A and a subset $B \subseteq A$, then B is the convex hull of A if the boundary through these points is convex and contains A.

There are many algorithms to calculate the convex hull for different use cases. But the number of points used in this application is so small that a simple algorithm and preexisting implementation was chosen [Wik18].

The selected algorithm is Andrew's monotone chain convex hull algorithm. It's runtime is $\mathcal{O}(n\log(n))$ and stems from the fact that a full sorting of the point list is needed.

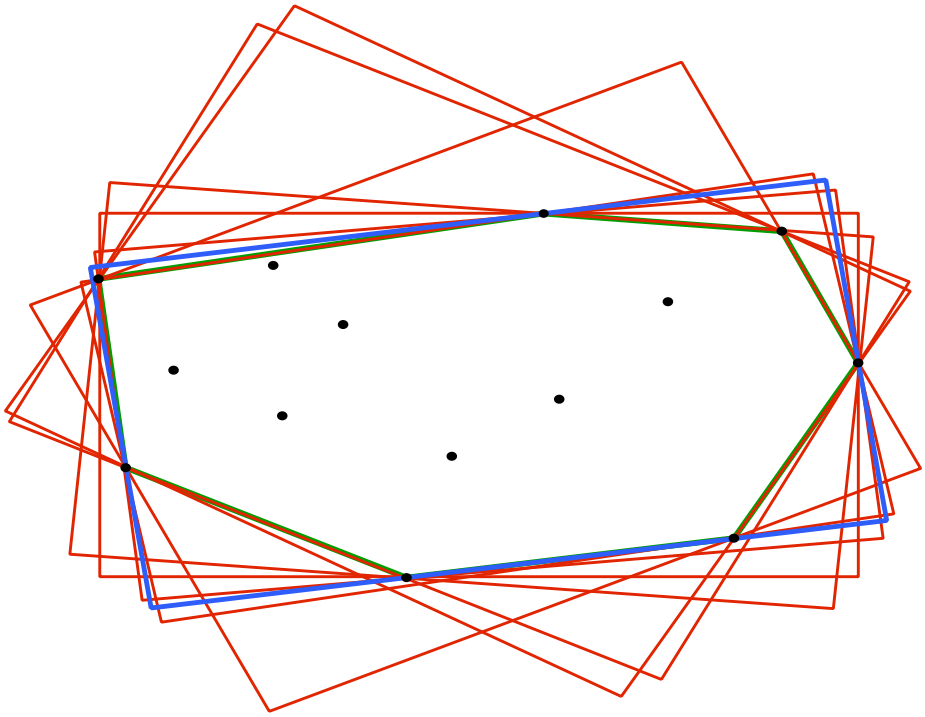


Figure 2.2: *A set of points. Convex hull in green, all evaluated boxes in red, oriented bounding box in blue.*

So the first step is to sort the list of points lexicographically first in x then in y direction. Afterwards the hull is constructed in two parts: the upper and lower half of the hull. To build the lower half of the hull the points are processed from left to right in lexicographically order. To build the other half the points are processed in reverse order.

The first two points are added to the hull's. It is tested whether the line through the last two hull points needs to be rotated clockwise or counterclockwise to intersect with the tested point. If a counterclockwise rotation is needed the point is not part of the convex hull and the next point is tested. But if a clockwise rotation is needed the points currently in the hull set needs to be revisited to determine which points are now not needed anymore for the hull. The process continues until all points are processed.

The convex hull of an example set can be seen in green in figure 2.2.

2.1.4 Oriented bounding box

The Oriented Bounding Box (OBB), sometimes also called minimal bounding box, of a set of points C, is a non axis-parallel rotated rectangle such that the enclosing area is minimal and still contains all the points of C.

As stated in [Tou83] at least one edge of the OBB has to be collinear with one of the edges of the convex hull of the points to be of minimal size.

To find the OBB of a given set of points the method of rotating calipers [Tou83] is used. See fig 2.2 for a graphical representation of the method.

The steps of the rotating calipers algorithm:

1. Calculate the convex hull of the points.
2. Calculate the minimum and maximum points with respect to the x and y coordinate.
3. Initialize an axis-parallel rectangle through the minimum and maximum points.
4. Loop over all edges of the convex hull.
   (a) Calculate the angle between the box edges and the adjacent convex hull edges.
   (b) Update the point from the edge with minimal angle to the point following the edge.
   (c) Recalculate the edges of the rectangle using the new set of points.
   (d) Calculate the area of the rectangle.
5. Select the rectangle with minimal area from the iterations.

2.1.5 Ellifit

Ellifit is a method described in 2013 by [PLQ13a] for fitting points to an ellipse. An ellipse has five different parameters: two for the center (x_c, y_c), two for the semi-axes (a, b) and one for the rotation (θ). The method uses an non-iterative least-squares approach to do a geometric fit of n points to an ellipse. It tries to minimize the distance from all given points to the nearest point on the ellipse.

The first step is to mean adjust all data points:

$$\forall x \in P : \tilde{x} = x - x_{mean} \tag{2.14}$$

$$\forall y \in P : \tilde{y} = y - y_{mean} \tag{2.15}$$

The actual fitting is done by solving the following matrix equation:

$$\begin{pmatrix} \vdots & \vdots & \vdots & \vdots & \vdots \\ \tilde{x}^2 & 2\tilde{x}\tilde{y} & -2\tilde{x} & -2\tilde{y} & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix} \phi = \begin{pmatrix} \vdots \\ -\tilde{y}^2 \\ \vdots \end{pmatrix} \tag{2.16}$$

The authors suggest to use the Penrose-Pseudo-Inverse $\bar{\Phi}$ calculated by

$$\bar{\Phi} = (\mathbf{X}^T \cdot \mathbf{X})^{-1} \cdot \mathbf{X}^T \cdot \bar{Y} \tag{2.17}$$

to solve the equation.

After fitting, the ellipse parameters need to be calculated from the solution ϕ. The mean adjusted center of the ellipse is given by the following equation:

$$\tilde{x}_c = \frac{\phi_3 - \phi_4\phi_2}{\phi_1 - \phi_2^2} \tag{2.18}$$

$$\tilde{y}_c = \frac{\phi_1\phi_4 - \phi_3\phi_2}{\phi_1 - \phi_2^2} \tag{2.19}$$

Calculate both semi-axes as well as the rotation of the ellipse:

$$a = \sqrt{\frac{2(\phi_5 + \tilde{y}_c^2 + \tilde{x}_c^2\phi_1 + 2\phi_2)}{(1+\phi_1) - \sqrt{(1-\phi_1)^2 + (4\phi_2^2)}}} \tag{2.20}$$

$$b = \sqrt{\frac{2(\phi_5 + \tilde{y}_c^2 + \tilde{x}_c^2\phi_1 + 2\phi_2)}{(1+\phi_1) + \sqrt{(1-\phi_1)^2 + (4\phi_2^2)}}} \tag{2.21}$$

$$\theta_c = -\frac{1}{2}\arctan(2\phi_2/(1-\phi_1)) \tag{2.22}$$

As a last step the center points are shifted into their original coordinate system by reverting the initial mean adjustment.

$$x_c = \tilde{x}_c + x_{mean} \tag{2.23}$$

$$y_c = \tilde{y}_c + y_{mean} \tag{2.24}$$

The equations presented here deviate from the original paper. The authors of the paper released the source code to their implementation, from which the algorithm above is reverse engineered. See [PLQ13b] for the implementation.

2.1.6 Fast and effective ellipse detection for embedded vision applications

The fast and effective ellipse detection for embedded vision applications [FPC14] is described as a method to find ellipses in arbitrary images. One of the focuses was to provide an algorithm that works on embedded devices. The authors used an older Samsung Galaxy S2 smartphone from 2011 for their tests.

The algorithm works in multiple filter stages. The stages and steps for each stage are depicted in figure 2.3.

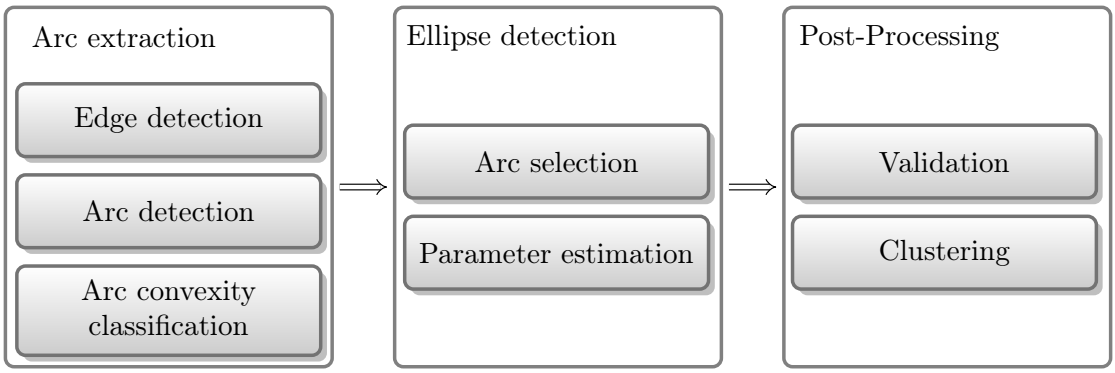


Figure 2.3: *The overall structure of the fast and efficient ellipse detection algorithm. There are three stages consisting of two to three steps.*

2.1.6.1 Arc extraction

The fist stage consists of three parts: the edge detection, arc detection and arc convexity classification. This stage is mostly directly working on the image. Its aim is to collect all connected edge pixels in the image that could be part of an ellipse.

Edge detection The edge detection is done with a modified canny edge detector with automatic thresholding. See the paper for more details. Every edge pixel is defined by its position and its phase $e_i = (x_i, y_i, \theta_i)$ [FPC14].

Arc detection For the arc detection the phase is divided into three regions $(+, -, 0)$:

$$\mathcal{D}(e_i) = \begin{cases} 0, & dx_i = 0 \wedge dy_i = 0 \\ +, & \operatorname{sgn}(dx_i) \cdot \operatorname{sgn}(dy_i) = +1 \\ -, & \operatorname{sgn}(dx_i) \cdot \operatorname{sgn}(dy_i) = -1 \end{cases} \tag{2.25}$$

with $\operatorname{sgn}(\tan(\theta_i)) = \operatorname{sgn}(dx) \cdot \operatorname{sgn}(dy)$
and dx, dy as derivatives of e_i in x, y direction

All pixels on the decision boundary with $\mathcal{D}(e_i) = 0$ are not considered for further processing.

All edge pixels that are connected and have the same $\mathcal{D}(e_i)$ form the arc α^k together:

$$\alpha^k = \{e_1^k, \ldots, e_{N^k}^k\} : \forall i, j : (\mathcal{D}(e_i^k) = \mathcal{D}(e_j^k) \wedge \text{Connected}(e_{i-1}^k, e_i^k)) \tag{2.26}$$

where N^k is the total number of points in the arc α^k
and $\text{Connected}(e_{i-1}^k, e_i^k)$ tests whether e_{i-1}^k, e_i^k are adjacent (8-connected)

8-connected means that both pixels share an edge or a corner, such that they are both one of the 8 neighbour pixels of each other.

The arcs found with equation 2.26 are collected in the set $\mathcal{R}_1$. To narrow down the number of arcs, the arcs with a low amount of edge points and a small oriented bounding box are selected into $\mathcal{R}_2$:

$$\mathcal{R}_2 = \{\alpha^k \mid N^k < Th_{length} \wedge \min(\text{OBB}(\alpha^k)_{width}, \text{OBB}(\alpha^k)_{height}) < Th_{obb}\} \tag{2.27}$$

with Th_{length} and Th_{obb} as thresholds

and $\text{OBB}(\alpha^k)$ to compute the OBB of arc α^k

Arc convexity classification Every arc in $\mathcal{R}_2$ will be classified by its convexity using $\mathcal{C}(\alpha^k) \in \{+,-\}$. To calculate the convexity of an arc the Bounding Box (BB) is used. The BB of an arc α^k is divided by the arc, such that there will be a region under (U^k) the arc and a region over (O^k) the arc. If the number of pixels under the arc is greater than the number of pixels over it then it is an upward arc($+$) otherwise a downward arc($-$).

$$\mathcal{C}(\alpha^k) = \begin{cases} + & = U^k > O^k \\ - & = U^k < O^k \end{cases} \tag{2.28}$$

A pseudo-code implementation for the convexity function is given in the paper: [FPC14, Alogrithm 1].

The goal of the convexity classification phase is to determine if the arc could be part of an upper right (I), upper left (II), lower left (III) or lower right (IV) edge of an ellipse. For this purpose the function $\mathcal{Q}(\alpha^k)$ is defined:

$$\mathcal{Q}(\alpha^k) = \begin{cases} I, & \mathcal{D}(\alpha^k) = + \quad \wedge \quad \mathcal{C}(\alpha^k) = + \\ II, & \mathcal{D}(\alpha^k) = - \quad \wedge \quad \mathcal{C}(\alpha^k) = + \\ III, & \mathcal{D}(\alpha^k) = + \quad \wedge \quad \mathcal{C}(\alpha^k) = - \\ IV, & \mathcal{D}(\alpha^k) = - \quad \wedge \quad \mathcal{C}(\alpha^k) = - \end{cases} \tag{2.29}$$

2.1.6.2 Ellipse detection

This stage consists of two parts: the arc selection and ellipse parameter estimation. The goal is to build 3-tuples of arcs that could possibly form three parts of an ellipse. The problem is that searching all possible combinations of three different arcs would be too time consuming.

To solve this problem the arcs are pre-filtered and collected into pairs of arcs in the arc selection step. Afterwards an ellipse center is estimated for each. With this estimate

all arc pairs are "clustered" into pairs of arc pairs such that they have one arc in common and the estimated centers are close enough.

The stages are explained in more detail in the next paragraphs.

Arc selection This step fits two arcs together into the arc pair $p^{ab} = (\alpha^a, \alpha^b)$. For a arc pair to be considered fitting ($\top$) the following mapping is used:

$$\mathcal{M}(p^{ab}) = \begin{cases} \top, & (\mathcal{Q}(\alpha^a), \mathcal{Q}(\alpha^b)) = (I, II) \quad \wedge (L_x^a \gtrsim R_x^b) \\ \top, & (\mathcal{Q}(\alpha^a), \mathcal{Q}(\alpha^b)) = (II, III) \quad \wedge (L_y^a \gtrsim L_y^b) \\ \top, & (\mathcal{Q}(\alpha^a), \mathcal{Q}(\alpha^b)) = (III, IV) \quad \wedge (R_x^a \lesssim L_x^b) \\ \top, & (\mathcal{Q}(\alpha^a), \mathcal{Q}(\alpha^b)) = (IV, I) \quad \wedge (R_y^a \lesssim R_y^b) \\ \bot, & \text{otherwise} \end{cases} \tag{2.30}$$

where L^a/R^a is the leftmost/rightmost edge pixel of arc α^a
and $\lesssim, \gtrsim$ are inequalities with tolerance Th_{pos}

For forming an ellipse both arcs must be from adjacent $\mathcal{Q}$-classes and they should not overlap much.

To get triples τ^{abc} out of the pairs all tuples are matched against each other with the following function:

$$\tau^{abc} = \begin{cases} \top, & |C^{ab} - C^{bc}|_2 < Th_{center} \\ \bot, & \text{otherwise} \end{cases} \tag{2.31}$$

where C^{xy} is the estimated center, calculated according to the following step.

Center estimation To estimate the center C^{ab} of a given arc pair p^{ab} a geometric property of the ellipse is used: *the midpoints of parallel chords are collinear* [YC94]. See fig. 2.4 for a sketch of this property.

A line through the midpoints of a set of parallel chords goes through the center of the ellipse. Thus the center can be found at the intersection of the lines belonging to two non-parallel chord-sets. But because of numerical instabilities and inaccuracies in the pixels this is a rather bad estimate.

The paper defines three points for each arc α^k the leftmost, middle and rightmost point (L^k, M^k, R^k). For an arc pair p^{ab} two chords are constructed: parallel to $\overline{L^a M^b}$ and parallel to $\overline{M^a L^b}$. The midpoints of the parallel chord to $\overline{L^a M^b}$ are named H_1^{ab} and these to $\overline{M^a L^b}$ are named H_2^{ab}. The lines l_1^{ab} and l_2^{ab} are intersecting H_1^{ab} and H_2^{ab}.

Because the midpoints are not very accurate the authors suggests to use the Theil-Sen estimator [Mat91, Algorithm 2] for a robust estimate of the slopes t_1^{ab} and t_2^{ab}.

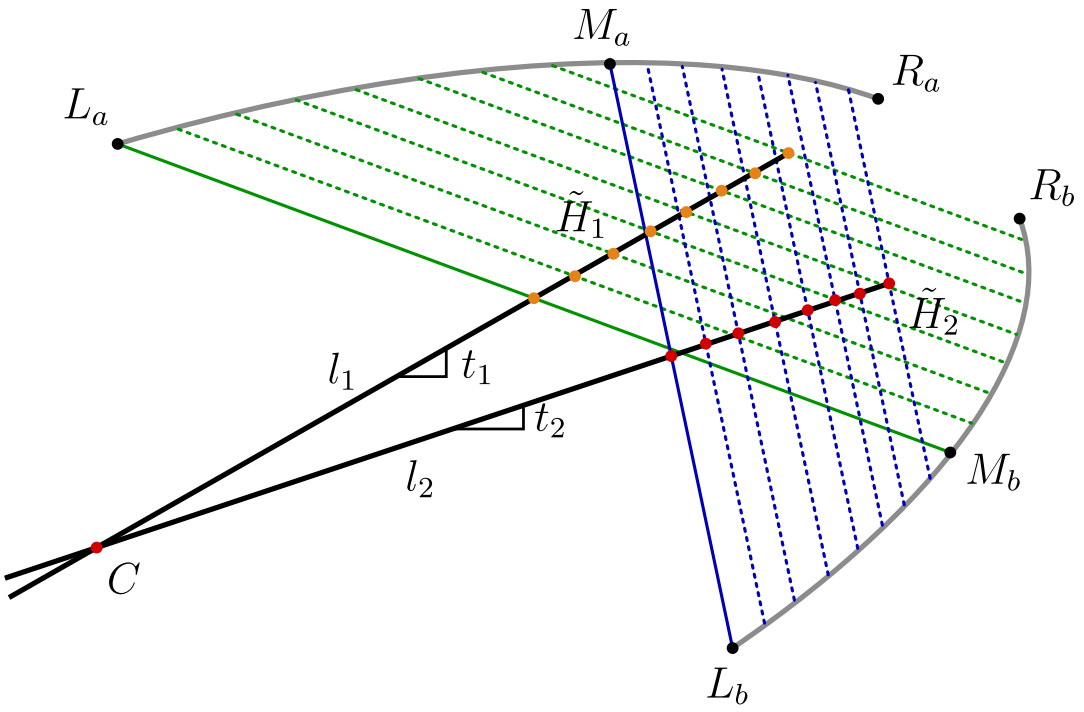


Figure 2.4: *The intersection of the midpoints of the parallel chords from one arc to the other meet at the center of the ellipse.*

The estimator calculates multiple slopes for different sets of points (eqn. 2.33). Thereafter the median of all calculated slopes is used as an estimate for the real slope.

$$
\begin{aligned}
& t_k^{ab} = \text{median}(S_k) && (2.32)\\
\text{with:}\quad & S_k = \{\frac{m_2.y - m_1.y}{m_2.x - m_1.x} \,|m_1 \in \mathcal{L}_k, m_2 \in \mathcal{U}_k\} \\
& \mathcal{L}_k = \{H_{k,1}, \ldots, H_{k,n_k/2}\}, \\
& \mathcal{U}_k = \{H_{k,n_k/2+1}, \ldots, H_{k,n_k}\} \\
\text{and}\quad & n_k = |H_k|
\end{aligned}
$$

To calculate the intersection of $l_k^{ab}, k \in \{1,2\}$ the median of H_k^{ab} as $\tilde{H}_k^{ab}$ is also needed. The center C is then calculated as the intersection of these two lines:

$$C.x = \frac{\tilde{H}_2.y - t_2\tilde{H}_2.x - \tilde{H}_1.y + t_1\tilde{H}_1.x}{t_2 - t_1} \quad (2.33)$$

$$C.y = \frac{t_1\tilde{H}_2.y - t_2\tilde{H}_1.y + t_1t_2(\tilde{H}_1.x - \tilde{H}_2.x)}{t_2 - t_1} \quad (2.34)$$

Ellipse parameter estimation The paper describes multiple methods to calculate ellipse parameters for τ^{abc}. They also develop their own simple method. But for the scope of this thesis the ellifit algorithm (section 2.1.5) is used. The five points L^a, R^a, L^b, R^b, L^c of the arc triplet τ^{abc} are used to perform the fitting.

2.1.6.3 Post-Processing

The post-processing step is the last stage of the algorithm. It is used to rate already found ellipses and tries to eliminate ellipses that represent essentially the same image contour.

Validation All ellipses that were fitted in the last steps are now validated with respect to all points belonging to the arcs. The paper suggest a simple scoring scheme to validate a ellipse fitting. Using the ellipse equation the "distance" of an arc point $(\bar{x}_i, \bar{y}_i)$ to the ellipse ε_i is calculated:

$$\begin{gathered} f(\bar{x}_i, \bar{y}_i, \varepsilon_i) = \\ \frac{(\cos(\varepsilon_i.\rho)(\bar{x}_i - \varepsilon_i.x) + \sin(\varepsilon_i.\rho)(\bar{y}_i - \varepsilon_i.y))^2}{\varepsilon_i.a^2} \\ + \\ \frac{(\sin(\varepsilon_i.\rho)(\bar{x}_i - \varepsilon_i.x) - \cos(\varepsilon_i.\rho)(\bar{y}_i - \varepsilon_i.y))^2}{\varepsilon_i.b^2} \end{gathered} \tag{2.35}$$

The value of the function is less, greater or equal to one, depending on the position $(\bar{x}_i, \bar{y}_i)$ of the point:

$$f(\bar{x}_i, \bar{y}_i, \varepsilon_i) = \begin{cases} = 1 \text{ ,if } (\bar{x}_i, \bar{y}_i) \text{ is on the boundary of } \varepsilon_i \\ > 1 \text{ ,if } (\bar{x}_i, \bar{y}_i) \text{ is outside the boundary of } \varepsilon_i \\ < 1 \text{ ,if } (\bar{x}_i, \bar{y}_i) \text{ is inside the boundary of } \varepsilon_i \end{cases} \tag{2.36}$$

With this function it is possible to define a set of points where the "distance" to the ellipse is less than 0.1. The ellipsoid region with $|(f(\bar{x}_i, \bar{y}_i, \varepsilon_i)| < 0.1$ is not equally thick (see fig. 2.5) everywhere. Distances measured with it are not directly comparable to distances of ellipses with other ellipse parameters. The percentage of the number of points near the boundary compared to the total number of points is used as a validation score:

$$\sigma = \frac{|\mathcal{B}|}{|\alpha^a| + |\alpha^b| + |\alpha^c|} \tag{2.37}$$

$$\text{where:} \quad \mathcal{B} = \{(\bar{x}_i, \bar{y}_i) : |f(\bar{x}_i, \bar{y}_i, \varepsilon_i) - 1| < 0.1\} \tag{2.38}$$

Ellipses with a score greater than $\sigma > Th_{score}$ are considered valid, all other ellipses are discarded.

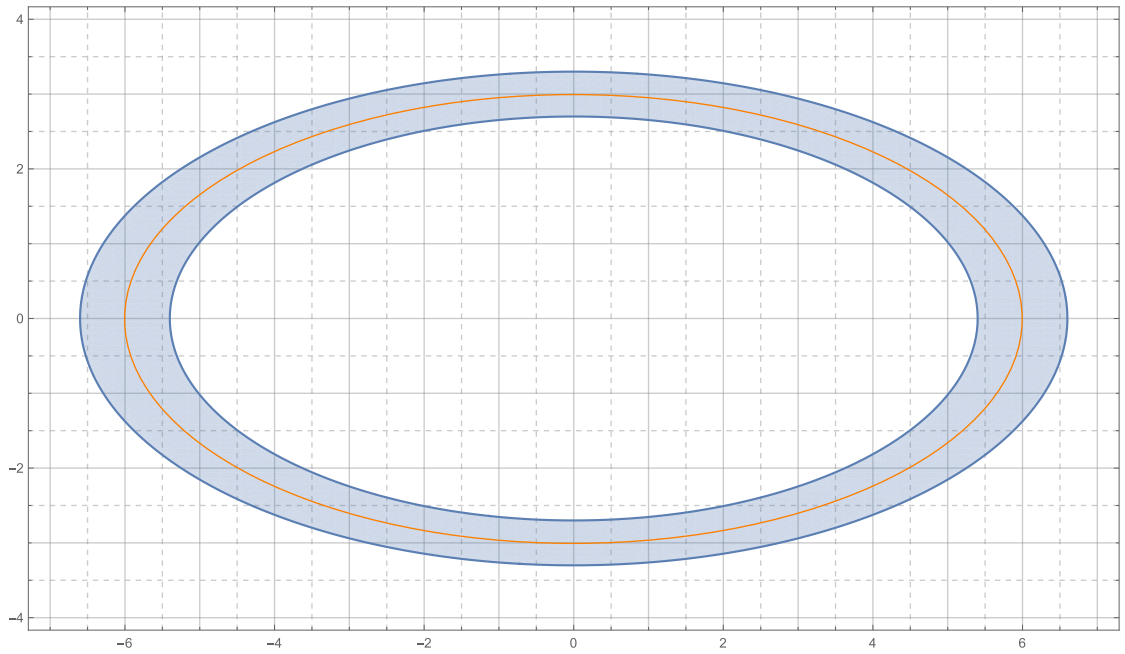


Figure 2.5: *The region $\mathcal{B}$ around the ellipse $\varepsilon = (0,0,6,3,0)$ (see equation 2.38). It can be seen that the boundary of the region is not equidistant to the orange ellipse everywhere.*

Clustering As a last step a clustering algorithm is used to get rid of duplicate ellipses. For two ellipses to be classified into the same cluster the function:

$$\Delta(\varepsilon_i, \varepsilon_j) = \delta_c \wedge \delta_a \wedge \delta_b \wedge \delta_\rho \tag{2.39}$$

$$\text{where:} \quad \delta_c = \sqrt{(\varepsilon_i.x - \varepsilon_j.x)^2 + (\varepsilon_i.y - \varepsilon_j.y)^2} < \min(\varepsilon_i.B, \varepsilon_j.B) \cdot 0.1 \tag{2.40}$$

$$\delta_a = (|\varepsilon_i.A - \varepsilon_j.A| / \max(\varepsilon_i.A, \varepsilon_j.A) < 0.1 \tag{2.41}$$

$$\delta_b = (|\varepsilon_i.B - \varepsilon_j.B| / \min(\varepsilon_i.B, \varepsilon_j.B) < 0.1 \tag{2.42}$$

$$\delta_\rho = \begin{cases} \frac{\angle(\varepsilon_i.\rho, \varepsilon_j.\rho, \pi/2)}{\pi} < 0.1, & \text{if } \left(\frac{\varepsilon_i.B}{\varepsilon_i.A} < 0.9\right) \wedge \left(\frac{\varepsilon_j.B}{\varepsilon_j.A} < 0.9\right) \\ \mathit{false}, & \text{otherwise} \end{cases} \tag{2.43}$$

$$\angle(x,y,r) = \begin{cases} y - x < -\frac{r}{2}, & (y-x) + r \\ y - x \geq \frac{r}{2}, & (y-x) - r \\ \mathit{otherwise}, & y - x \end{cases} \tag{2.44}$$

is used. The equivalence metric is a slightly modified scheme of [PL10].

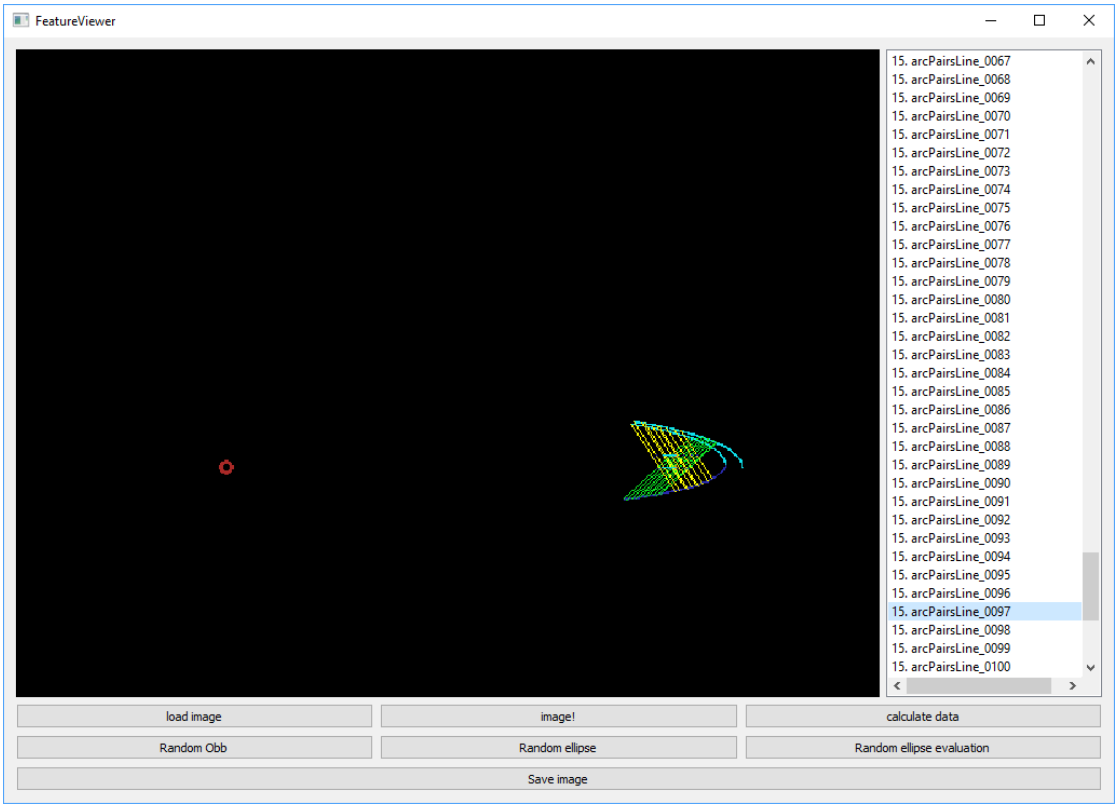


Figure 2.6: *The GUI application build for prototyping. It shows an image of the center estimation stage.*

All ellipses are sorted with respect to their score and compared with each existing cluster using equation 2.39. If it matches that cluster it will be discarded, otherwise it will be added as a new cluster.

The remaining clusters are the ellipses found by the algorithm.

2.2 Used software

This section explains the software used for making the experiments. In section 2.2.2 a short introduction, to the HULKs NAO framework given. The following section describes the Intel VTune Amplifier sampling profiler, how it works and how it was integrated into the NAO framework.

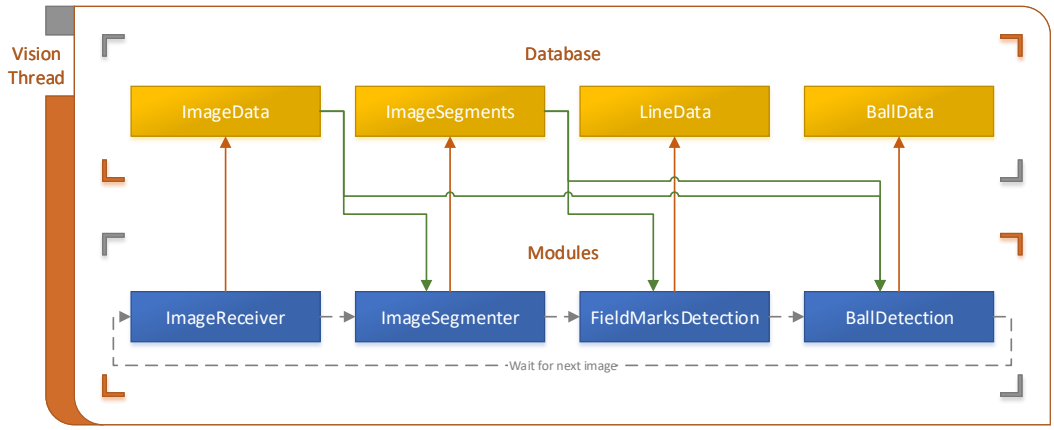


Figure 2.7: *A small example how the runtime order of modules in the vision thread could look like. Modules are ordered such that every dependency (orange arrows) is produced before another module needs it (green arrows). The cycles of all modules are then executed from left to right. After all modules are finished it waits for a new image and rerun every module cycle to update the products.*

2.2.1 Feature viewer

To be able to implement and test the algorithm stages a C++ application using Qt [Qt 17, Version 5.9] and OpenCV [Bra00, Version 3.2.0] was build. The GUI features a list of images to for visualizing different stages of the implemented alogrithms. It also has an image viewer and a couple of buttons. See figure 2.6 for an overview.

2.2.2 HULKs NAO Framework

The NAO framework developed by the HULKs SPL team [HUL17] is the software framework that is used to develop the software for the RoboCup competition. The framework models a data driven directed data flow graph. It is composed of relatively small modules that each have dependencies and productions that are simple structures with all needed data. When the software is started it has a list of modules that should be run. It can inspect which dependencies and productions a module requires. With this information it tries to find an execution order such that every dependency is produced from another module before it gets executed.

After this step all modules are executed. Every module has a *cycle* method. The purpose of this method is to take the current dependencies and transform them into its productions. All module *cycle* methods are called in the correct order because of the ordering. After all modules are finished the code waits for the next input (an image or

joint data) and all modules are rerun again.

The framework is designed to be able to easily integrate new team members. To reach this goal it tries to reduce the amount of code a new member needs to keep in mind in order to be able to understand a module. To understand a module a member only needs to look at the *cycle* method, the incoming dependencies and the structure that is produced from the module.

2.2.3 Intel VTune Amplifier 2018

Intel VTune Amplifier [Int17] is a sampling profiler. With it, it is possible to collect properties of another program. The profiler starts the profilee, preempting it a configurable amount of times per second and collects data about its current state. After the application finished the profiler writes all collected data to persistent storage for later inspection. VTune can then shows how often every function inside the application was called or how long its execution took. With VTune it is also possible to get an estimate of the impact of every assembly instruction on the runtime of the program.

VTune also has an API that can be called from inside the profilee to get a better structured overview of functions that got called per region of the application. For this thesis the NAO framework was modified to be able to mark the start and end of every module cycle. With these modifications it was easy to test and compare different implementations of the algorithms. It was also used to find hot paths in the algorithms and optimize them.

Chapter 3

Implementation

This chapter is about the implementation and optimizations of the algorithms presented in chapter 2. At first the implementation was prototyped in the "feature viewer" from section 2.2.1 with the help of OpenCV. Afterwards the algorithm was ported to the HULKs framework to analyze the runtime of the implementation on the real hardware. The goal was to check if it is even possible to run the algorithm in real-time on this limited hardware.

3.1 Sobel edge detection

Sobel edge detection is a important part of many state of the art computer vision algorithms. To maximize the reusability the edge detection was implemented in its own "SobelImageProvider" module.

The output of this module is an interleaved two channel 8-bit image. The first channel is the gradient and the second is used for the $\mathcal{D}$-phase of the fast and efficient ellipse detector (see 2.25).

The SPL team Nao Devils implemented the gradient part of the sobel edge detection using highly optimized Streaming SIMD Extensions (SSE) intrinsics [Alb+16]. SSE is a CPU extension that is used to do the same calculations in parallel on multiple data values in one CPU-cycle. This is really useful to further optimize the throughput and utilization of a modern x86 CPU.

The goal of using SSE here is to speed up the calculation of the sobel operation on the whole image. The incoming image is a YCbCr422 encoded image. In YCbCr422 the chroma components Cb, Cr are sampled at half the sample rate of the luma channel Y. The memory layout is $(Y, Cb, Y, Cr, \ldots)$.

With SSE it is possible to load 64-bits of data into *xmm* register and do the same calculations on independent parts of these data. For example a 128-bit *xmm* register can

be split into 16 8-bit registers. The *_mm_adds_epu8* instruction takes two *xmm* registers, treats them as 16 8-bit registers and adds all of them together independently.

The *addS* instruction is a saturated addition (no wrap-around). For example while calculating the amplitude of the gradient is is more helpful to stay at the maximum 8-bit value than to wrap-around to the smallest possible value and not be able to find this edge anymore.

3.1.1 Implementation details

The following part gives a short overview of the inner workings of the implementation. For all details please read [Alb+16, P. 9].

The convolution is only calculated for the Y channel of the image but the data is interleaved in memory. Loading 256-bits of data into two *xmm* registers and using the *unpack* instruction six times reorders the Y channel data in the first register. To be able to calculate the convolution, data from three adjacent rows is needed. 16 Y values of three rows gets loaded and unpacked into three registers.

From the sobel kernels (see equation 2.8):

$$M_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \qquad M_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

it can be seen, that six different values with different factors from different columns are needed to calculate the derivative of one pixel. The solution domain is $-4 \cdot 255$ to $4 \cdot 255$. But an 8-bit register can only hold 255 values. To get around this problem the calculations get split into a negative and positive result, adding/subtracting only the positive/negative counted values. Also the whole equation is divided by four such that the factors end up as 1/2 and 1/4. Thus the different rows need to be divided by two and four (right shift) and the subregisters need to be shifted by one and two positions to the left. Because of the shifting is is only possible to calculate 14 values in one loop iteration instead of the full 16 vector elements available in hardware.

After this step the corresponding registers only need to be added and subtracted in the right way to calculate 14 results. To finally calculate the gradient (see equation 2.12) a approximation is used:

$$\sqrt{G_x^2 + G_y^2} \approx \alpha \max(G_x, G_y) + \beta \min(G_x, G_y) \tag{3.1}$$

with $\alpha = 1, \beta = 0.25$. The multiplication can be easily performed by a right shift operation. For further details see [Alb+16, P. 14].

Now the data of the next pixels are loaded and the steps are repeated.

3.1.2 Optimization

After embedding the implementation into the HULKs framework a profile run was made to analyze the runtime and find possible points for further optimization.

Two points were found for good optimization opportunities:

Image boundary After the whole image is processed, all pixels on the outer image boundary are set to zero. This is like a second pass over essentially the whole memory area. A sizeable part of the total runtime can be attributed to this operation. Since this behavior was not needed a new template parameter is introduced to disable this second pass.

Branch prediction The function has parameters to determine if the result should be the x derivative, y derivative or the gradient. The code has multiple *if* statements to switch between these different cases. Looking at the assembly code of the function and the calculated impact on the runtime, all *if* statements had a great impact on the runtime. To guide the compiler to optimize, these jumps away all parameters known at compile-time were moved into a *template parameter*. But this alone did not help to let the compiler optimize the branches away. To be able to finally remove them the C++17 feature *if constexpr* had to be used. To explicitly instruct the compiler to evaluate the condition at compile time. Afterwards all branches were gone.

3.1.3 Extension to the implementation

Since the implementation only calculates the gradient and the fast and efficient ellipse detection method also needs the $\mathcal{D}$-phase, its calculation was directly embedded into the sobel implementation with SSE intrinsics. As a reminder the equation 2.25 to calculate the phase is:

$$\mathcal{D}(e_i) = \operatorname{sgn}(dx_i) \cdot \operatorname{sgn}(dy_i) \tag{3.2}$$

The first part is to calculate the sgn. This function equals -1 if the input is below zero and 1 if it is above. dx_i and dy_i is calculated by adding the negative parts to the positive parts. Exploiting the already split up negative and positive parts, it is sufficient to compare their values to get the sign of the final addition. This can be implemented with the compare and maximum instructions *_mm_cmpeq_epi8* and *_mm_max_epu8*. First get the maximum of the negative/positive part then compare that to the positive part.

To put this into pseudo-code:

```
px = max(gx_neg, gx_pos) == gx_pos ? +1 : 0;
py = max(gy_neg, gy_pos) == gy_pos ? +1 : 0;
```

This can be used to calculate the sign of the x and y derivative. But it does not handle the special case with dx or dx to be 0 as described in 2.1.6.1. But this is not a big problem.

Afterwards the two values needs to be multiplied. But instead of multiplication it is much easier to also use the *cmpeq* instruction. If *px* and *py* are equal the result is $+1$ if they are not equal the solution is -1. Because this information only requires 1-bit the other 7-bits could be used for other information (see section 3.4.1.1). To clear all bits but the highest the result of the compare is anded with $0x80$.

As a last step the solution needs to be interleaved with the gradients. To do that the *unpack* instruction can be used. Resulting in nine additional instructions per 14 pixels.

3.2 OBB Implementation

The general idea of data structures and implementation for this algorithm were taken from [Gei14].

To calculate the OBB the convex hull of the points is needed. A working implementation of Andrew's monotonic chain algorithm can be found in WikiBooks (see [Wik18]). It was modernized according to C++11 and adapted to the use of OpenCV for the use in the feature viewer.

3.2.1 Implementation

After calculating the points of the convex hull they are augmented with the normalized direction of the corresponding edge. For the initialization, the algorithm also needs the minimal and maximal points with respect to x and y (four points in total). Temporary variables for the current left, right, top and bottom points and edge direction are created and initialized with the min/max points and $(0,1),(1,0),(0,-1),(-1,0)$ to form the BB of all points.

The algorithm needs a number of iterations equal to the number of points in the convex hull. For every iteration step the angles from the augmented edge direction of the current points to the current rectangle edge directions are calculated. The point with the minimal angle gets updated to its next point in the convex hull (clockwise). Now also all edge directions need to be updated to be parallel/orthogonal to the new edge.

To calculate the area of this rectangle their corner points need to be determined via the intersection of the four lines (points + edge directions).

The area of the new rectangle is compared to the current best one. If it is better update the best one and do the next iteration step.

3.2.2 Optimization

The original implementation of the algorithm offers a optimization opportunity. It uses $\arccos(\langle \mathbf{a}, \mathbf{b} \rangle)$ to calculate the angles between the edges and convex hull. The arccos is a very CPU intensive function and does not need to be calculated here. $\mathbf{a}$ and $\mathbf{b}$ are already

normalized and arccos is a monotonic function in $\{-1,1\}$ so maximizing for $\langle \mathbf{a}, \mathbf{b} \rangle$ is sufficient.

3.3 Ellifit

The authors of ellifit published an OpenCV and a matlab implementation of their code in [PLQ13b]. The code was also ported to the Eigen3 because the HULKs framework does not use OpenCV.

3.3.1 Optimization

The paper suggests to calculate the Moore Penrose pseudo-inverse to solve the system of linear equations. Calculating the inverse of a matrix is rather expensive and often not needed. The inverse can be calculated computationally simpler by using the singular value decomposition. But an even better way to calculate the solution to the equation is to use QR-decomposition and back substitution. It is much faster to calculate and reduces numerical errors see [Bro+12, p. 973f].

$$\text{minimize } \left\| \bar{Y} - \mathbf{X} \cdot \bar{\Phi} \right\|_2 \tag{3.3}$$

$$\mathbf{X} = QR \tag{3.4}$$

$$R\bar{\Phi} = Q^T \bar{Y} \tag{3.5}$$

Q is an orthogonal matrix R is a upper triangular matrix

Φ can now easily be calculated using back-substitution. Eigen3 has a simple function for this:

```
auto qr = X.householderQr();
auto ph = qr.solve(Y);
```

3.4 Fast and effective ellipse detection

One of the biggest problem with this paper was that the authors did not publish an implementation of the method. While most of the paper is pretty straight forward mathematically, at least some parts had no clear path to an efficient implementation. Another interesting question for some parts is the selection of data structures for best efficiency.

3.4.1 Implementation details

Since the algorithm was prototyped using OpenCV the canny edge detection and oriented bounding box implementations of OpenCV were used. But since the HULKs framework does not use any OpenCV, all algorithms had to be implemented for the NAO. Also in some cases the OpenCV implementations of the algorithms are not optimal.

The canny edge detection was reviewed in a bachelor thesis of another team member and was deemed too slow. Because of this and an already existing fast sobel edge detection implementation using SSE, this thesis settled on changing the canny edge detection to only use the sobel stage.

3.4.1.1 Arc extraction

This was the part that was left most unclear in terms of implementation. Trying to implement it there were already two major questions:

- Which data structure should be used to represent an extracted arc?
- How to implement the extraction with minimal number of memory accesses?

Data structures After looking into chain coding schemes such as [WH08] it was decided that it was not worth using them because it made calculation of the OBB much harder and the arcs have a very small area so it would not save much memory space. The natural choice was to use a linear list of arcs. The data structure representing an arc includes the list of edge points, the border class ($\mathcal{D}$-phase), the convexity class, the $\mathcal{Q}$-class and the bounding box.

Arc following The first attempt of finding all arcs in the canny image was to linearly scan the whole image left to right, horizon to bottom and search for edge points. For every edge point, search if it is connected to an already found arc. If it is not connected add a new arc to the list. Checking if a point is connected to an arc requires to test every pixel in the arc if it is connected. This is rather slow and scales badly with many edge points on an arc.

In a second iteration it was tried to overcome this issue by calculating the bounding box whenever a new pixel was added to an arc. Checking if a point is connected to the arc could now be simplified in many cases. Check first if the pixel is inside the BB, if not continue, otherwise check every pixel associated with the arc if it is connected to the new edge point. But in many cases the loop over all edge pixels is still needed.

The last iteration used one of the seven free bits in the second channel of the sobel image (see section 3.1) to mark if a pixel was already visited. This had a big impact on the runtime because it is not needed anymore to check if a pixel was already added to the arc point vector.

As discussed in section 3.4.1 the canny edge detection was changed to a use the simpler sobel edge detection. This introduced two new problems for the arc extraction. An edge is not simply bidirectional anymore. It could extend in several directions from one point. Also the image is no longer binary.

To produce a binary image a threshold was introduced at which point a pixel was detected as an edge. To solve the other problem a recursive edge following algorithm was introduced:

Find a pixel that is considered an edge. Search all 8-connected pixels if they are an edge pixel. Push all edge pixels on a stack. Pop a pixel from the stack add it to the arc and follow/push all its edge pixels. Continue processing pixels from the stack until there are no connected pixels anymore.

After this stage all found arcs are stored in a list of arcs. Each arc is represented by a list of all its associated pixels.

Arc convexity After an arcs is extracted its convexity is calculated according to the pseudo-code given in the paper. Then the $\mathcal{Q}$-class is calculated using the convexity and the border class (see equation 2.29).

Arc filtering After all arcs are extracted they are filtered according to equation 2.27. Using the sobel edge detection made the short side of the OBB wider. Because of this, straight lines are often not rejected anymore.

To cover that case a new filter was added. It is based on how much of the OBB area is covered by edge pixels. If it is higher than 70 % then the arc is rejected.

3.4.1.2 Ellipse detection

Collect arc pairs To find all matching arc pairs according to equation 2.30 just iterate for every arc over every other arc and check the condition. If both of them match, store a tuple inside a list with pointers to both arcs and the center that will be calculated in the next step.

Collecting parallel chords To estimate the center of the collected arc pairs, a procedure to calculate parallel chords from arc to arc needed to be implemented. The authors of the paper did not share their approach to calculate these. The first step is rather easy: chose one outer point on one arc and the midpoint on the other and calculate the line through both points. Afterwards find parallel lines bound by the two arcs.

To calculate parallel chords there are in principle two ways:

Calculate the slope of the line, move it in normal direction and calculate the intersections with both arcs.

Or iterate through the edge points of both arcs and calculate the slope for the line through both and if the slope is equal to the original add it to the list of parallel chords.

The first method has the problem that calculating the intersection means to iterate through all arc points to calculate the distance to the line. The process of the second approach can easily be handled more locally. Advance the iterator of one arc forward a few edge points and move the iterator of the other arc points with a small step size forward, searching for a good parallel line on its way.

After a parallel chord is found, calculate the midpoint and add it to a list.

Center estimation The paper suggest to estimate the center lines using the Theil-Sen estimator. This technique divides the midpoints in two parts and calculate the slope of multiple lines through pairs of the two subsets. Afterwards the median of all slopes is used as an estimate for the slope. The median is also used on the x and y positions of the midpoints to calculate a robust estimate of a point on the center line.

While testing the implementation of this, it was found that there were multiple cases were the center lines were close to infinity because they were parallel to the y axis. To mitigate this problem the algorithm was changed to calculate the direction vector of the center line. The median of these vectors was calculated according to the angle of the vectors. But angles are values inside a quotient ring and it is not possible to sort the values and take the value in the center.

With taking the center values in a quotient ring modulo 360 the values: {0, 1, 2, 358, 359} would end up with a median of 358, but the natural choice of values would be 0 because the ring wraps around at the end and a better sorting would be {358, 359, 0, 1, 2}. So a look at calculating the median in a quotient ring is needed.

Calculating the median in a quotient ring Looking at the definition of the median it is possible to derive an algorithm to calculate the median in a quotient ring:

> The median value $\tilde{x}$ for an even number of members in the set $\mathcal{M}$ is minimizing the sum of absolute errors to all values:
>
> $$\forall x \in \mathcal{M} \quad \exists \tilde{x} : \sum_{i=1}^{n} \angle(\tilde{x}, x_i) \leq \sum_{i=1}^{n} \angle(x, x_i) \tag{3.6}$$
>
> if $x \in \mathbb{R} : \angle(x,y) = |x-y|$
>
> if $x \in \mathbb{R}/\pi\mathbb{Z} : \angle(x,y) = \angle(x,y,\pi)$ from eqn. 2.44
>
> Compare with [Kog13, Median of circular values] for more information.

Calculating the error for every element in the set using the minimal walk between the two using wraparound and select the value with the minimal error yields the median value of this set.

Collect arc triplets To find appropriate triplets iterate over all arc pairs. For every pair of pairs test whether they have one arc in common and three different $\mathcal{Q}$-classifications. If they have, calculate the distance of both estimated centers. If the distance is less than a threshold calculate the ellipse parameters using ellifit.

Ellipse parameter estimation To estimate the ellipse parameters of the arc triplets use five points of all tree arcs and use ellifit (section 2.1.5). Afterwards calculate the fitting score of the ellipse. Discard the ellipse if the score is less than a threshold. Otherwise append it to a list together with the score.

3.4.1.3 Post-Processing

Ellipse validation To calculate the score of a given ellipse iterate through all arc points and calculate the distance to the fitted ellipse using equation 2.35. Save the number of points inside the defined ellipse boundary corridor. Return the quotient 2.37 as the score value.

Ellipse clustering Sort all remaining ellipses according to their score. Start with the ellipse with the highest score and add it as a new ellipse cluster. Iterate over all other ellipses. If they can be assigned to an existing cluster using equation 2.39 discard them else add as a new cluster.

3.4.2 Optimization

This chapter will summarize all changes made to the original paper:

The sobel edge detection is used instead of the canny detection because of runtime issues. Also there was a preexisting fast implementation of the sobel detection mechanism that made it more feasible to reach the runtime goal.

The authors of the paper suggests to calculate the slope of the center lines for an estimation of the center. But if the direction is near to 90° the slope will be close to infinity and thus the calculation fails numerically. To mitigate the problem the angle of the line is calculated instead. But now the Theil-Sen estimator cannot be used anymore. To calculate the median of the estimated angles a new algorithm was implemented.

The authors suggests to use their method of ellipse parameter estimation but this implementation only uses the ellifit method.

This variant does not implement the averaging of the ellipse parameters per ellipse clusters because once the correct arc triplets are found the estimate of the ellipse parameters are already good enough.

Chapter 4

Evaluation and Discussion

This chapter starts with an analysis of the different ellipse parameters that can be encountered by a robot. Afterwards the performance of the presented algorithm is analyzed. The dataset used for verification is explained in the following section. Then the results of the algorithm on the dataset are shown and the runtime performance on the real robot is tested.

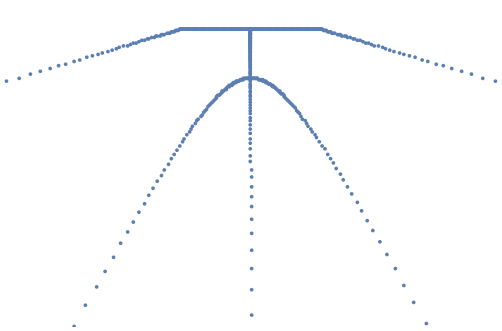
4.1 Reverse center circle projection

To get a better understanding of the ellipses the robot could actually encounter, a mathematical study was performed. The reverse projection from section 2.1.1 was used to calculate the image a robot would see if it was placed on a specific position on the field.

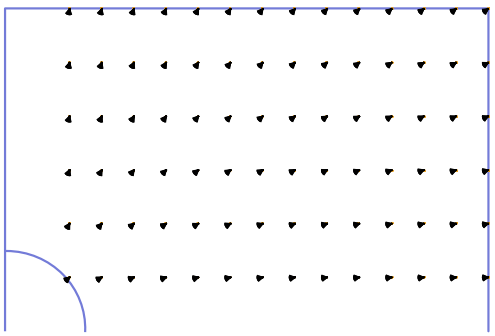
Mathematica notebook To be able to quickly analyze the data, a small Mathematica[Wol17] notebook was written. It contained the forward and backward projection equations for a NAO standing in the ready pose. Also the ellifit algorithm was implemented to be able to get the ellipse parameters of the projected center circles.

The procedure is implemented according to:

1. Calculate points on the center circle.
2. Calculate multiple positions on the field.
3. For all positions translate the circle points relative to it.
4. Use the reverse projection to project the circle points into an image.
5. Use ellifit to calculate the parameters of the ellipse.



(a) *An image of the projected center circle from within, seen with a very big wide angle lens. It can be seen that this is not fitable to a ellipse.*



(b) *All considered positions with the viewing angles printed on a field map. Only a quarter of the field needs to be considered because of the field symmetry.*

Figure 4.1: *Projection from inside the center circle and the different evaluated positions on the field.*

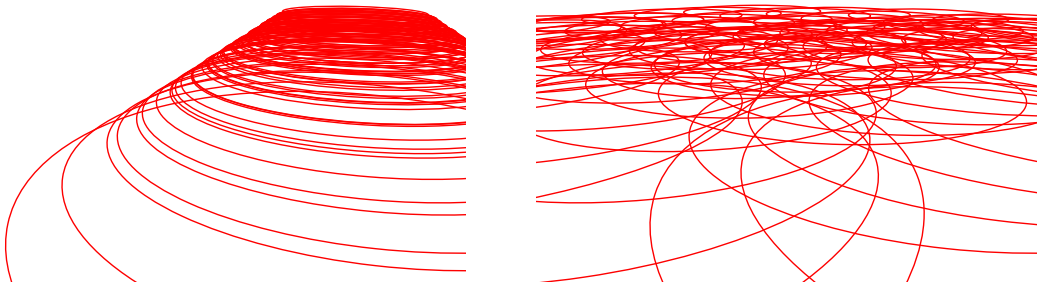
Projections within the center circle Using this scheme an interesting observation was made: If the robot is inside the center circle the projection of it is not an ellipse, see figure 4.1a. Mathematically the perspective projection of a circle will generally result in a conic and only one possible configuration of it is an ellipse. The other ones are parabolas and hyperbolas. So using the algorithm presented in this thesis will only work if the NAO is outside the center circle. In this case the projection is in fact an ellipse.

In this case of figure 4.1a the projection is in fact a hyperbola which could be seen if the circle points in the back of the robot would also be mathematically projected into the image.

Evaluated positions The positions are evenly distributed over a quarter of the field, see figure 4.1b. A quarter of a field is enough, because of the symmetry of the field. All positions were calculated with a viewing angle of $-30°$ to $30°$ to the middle of the center circle. Figure 4.2b shows every 7^{th} ellipse of the dataset in one image. To get a better understanding of possible ellipses figure 4.2a shows all considered ellipses but only with a $15°$ rotation of the robot to the middle of the center circle.

Evaluation Figure 4.3 features a box plot related to the distribution of ellipse parameters encountered by a NAO. As expected some configurations are much more likely than others.

One notable fact is that most of the time the ellipses are much wider than they are high. Because the robot looked above the horizon the y coordinate is not limited by the maximum image height but by the horizon. Another interesting observation is that



(a) *Many possible circles seen from different positions on the field with a rotation of* 15° *to the circle center.*

(b) *Many possible circles seen from different positions on the field but with* −30° *to* 30° *rotation to the center of the circle.*

Figure 4.2: *Projected center circles from different positions.*

the possible rotations of the ellipses are limited. The maximal encountered rotation of ellipses were ±20°.

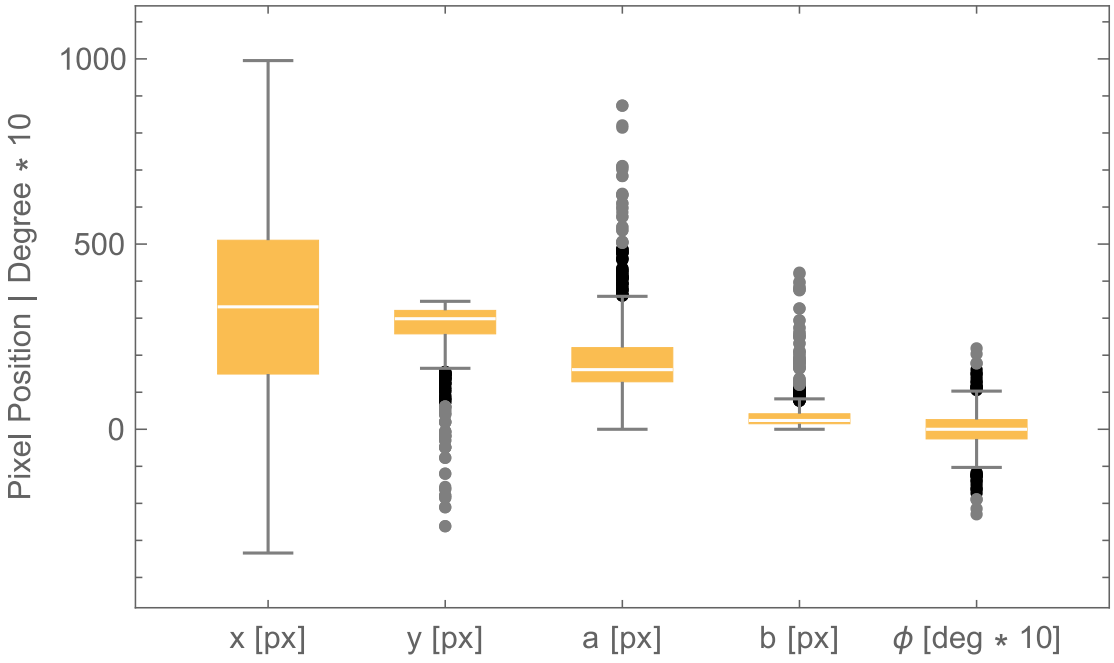


Figure 4.3: *The distribution of ellipse parameters after projecting the center circle from many different positions and angles. All positions are outside of the center circle.*

4.2 Dataset and validation

This section discusses the preparation of the verification dataset and the validation of the results.

Dataset To verify and test the presented method a set of 1000 artificially generated images containing only ellipses were used. The parameters of the ellipses are sampled from a normal distribution for each parameter modeled around the distribution presented in figure 4.3.

After generating a possible parameter set a few conditions were checked to accept the ellipse as valid. The primary semi-axis had to be bigger than the secondary one. The primary semi-axis was limited to min 86 and max 880. The secondary one was limited to min 8 and max 330 and more than 50 % of the ellipse had to be visible on the image. This way it should be possible for an algorithm to have enough information to be able to find the ellipse.

Validation To be able to validate the fitted ellipses a slightly modified variant of equation 2.39 is used as a performance metric:

$$\begin{aligned}\mathcal{M}(e_1,e_2) = 1 - \\ \sqrt{(\varepsilon_1.x-\varepsilon_2.x)^2+(\varepsilon_1.y-\varepsilon_2.y)^2}/\min(\varepsilon_1.A,\varepsilon_1.B,\varepsilon_2.A,\varepsilon_2.B) - \\ |\varepsilon_1.A-\varepsilon_2.A|/\max(\varepsilon_1.A,\varepsilon_2.A) - \\ |\varepsilon_1.B-\varepsilon_2.B|/\min(\varepsilon_1.B,\varepsilon_2.B) - \\ (\varepsilon_1.\rho-\varepsilon_2.\rho)/\pi\end{aligned} \tag{4.1}$$

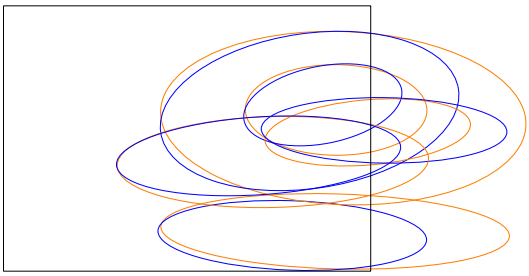
This way each experiment gets a score of how well the ellipse was fitted to the incoming data.

4.3 Results

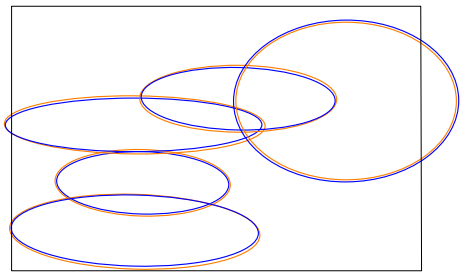
This section presents the results of the implemented algorithms against the generated dataset.

Randomly generated ellipses After evaluating the 1000 randomly generated images the algorithm found an ellipse in roughly half of the images. In the other half no ellipses could be found. But the ellipses that were found were of a very good quality.

See figure 4.4 to see the best and the worst fitted matching ellipses from the set. Most of the bad matching ellipses have a big part of the contour outside the image. Because



(a) *The five ellipses with the worst fitted ellipses. The box is the viewport of the camera. Because there is a part of the ellipse missing the performance is really bad.*



(b) *The five ellipses with the best fitted ellipses. The box is the viewport of the camera. There is not much difference between the original and fitted ellipses.*

Figure 4.4: *Images of the five best and worst fitted ellipses. Original ellipses in orange, fitted ones in blue.*

this part of the information is missing the algorithm has a hard time calculating the semi-axis of the ellipses.

The first box in figure 4.5 represents the distribution of the internally calculated scores from the validation section 2.1.6.3.

The second one shows the validation score from the best found ellipse cluster to the real generated ellipse parameters. This score is calculated using a variant of eqn. 2.39:

$$\langle \varepsilon_i, \varepsilon_j \rangle = 1 - (\delta_c + \delta_a + \delta_b + \delta_\rho) \tag{4.2}$$

$$\text{where:} \quad \delta_c = \frac{\sqrt{(\varepsilon_i.x - \varepsilon_j.x)^2 + (\varepsilon_i.y - \varepsilon_j.y)^2}}{\min(\varepsilon_i.B, \varepsilon_j.B)} \tag{4.3}$$

$$\delta_a = \frac{|\varepsilon_i.A - \varepsilon_j.A|}{\max(\varepsilon_i.A, \varepsilon_j.A)} \tag{4.4}$$

$$\delta_b = \frac{|\varepsilon_i.B - \varepsilon_j.B|}{\min(\varepsilon_i.B, \varepsilon_j.B)} \tag{4.5}$$

$$\delta_\rho = \frac{|\varepsilon_i.\rho - \varepsilon_j.\rho|}{\pi} \tag{4.6}$$

It can be seen, that the algorithm is able to mostly reach a score of 80 % to 100 % but the real score is much lower in most cases. This could be due to the fact that it happens very often that the algorithm matches arcs from the outer boundary to ones from the inner boundary. This obviously leads to a bad fitting of the ellipses.

The last box shows the distribution of the distance of the best centers from the pairwise center estimation step to the real center. It can be seen, that most of the time the estimate is rather accurate. But it can also fail very badly.

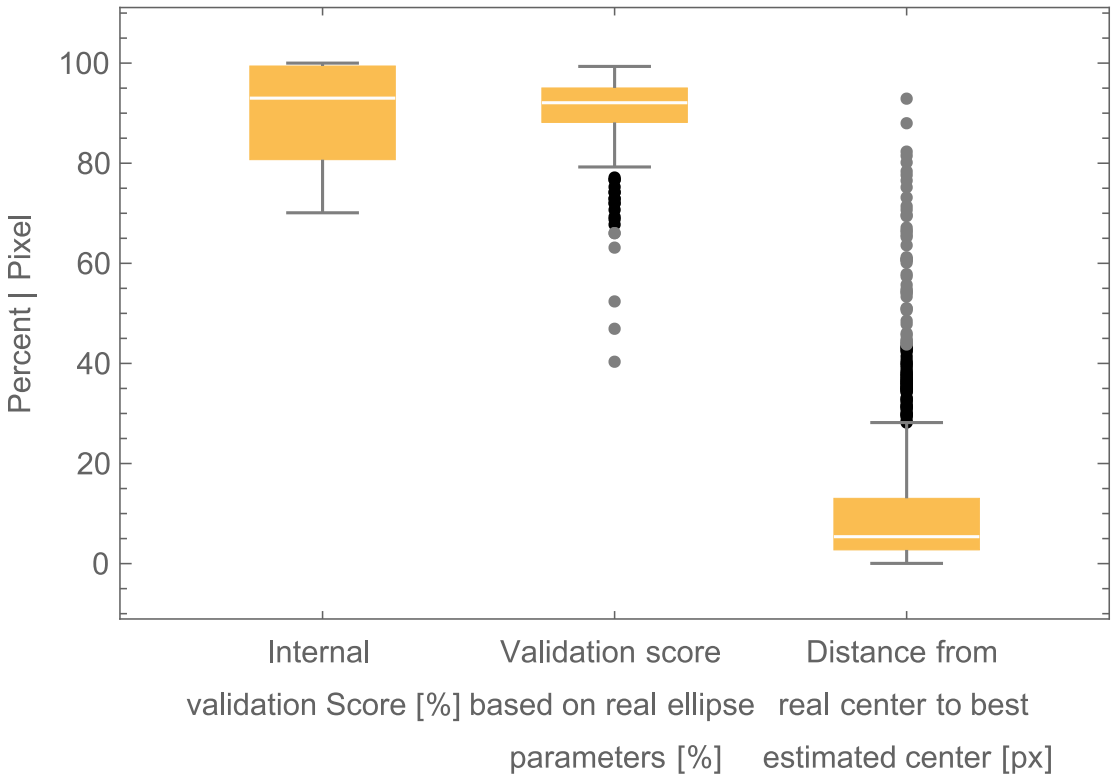


Figure 4.5: *Two different metrics for the final fitted ellipses and the distance of the best estimated center circle to the real one.*

Runtime performance Because the images are recorded with a frequency of 60 Hz the whole vision cycle can last 16 ms at maximum. But the motion cycle and other processes also need CPU time. In practice the calculations should last no longer than 10 ms.

The original SSE sobel implementation from the NAO Devils takes about 3.45 ms per image. After the "constexpr" and border clearing optimization it is about 0.5 ms faster. But after adding the $\mathcal{D}$-Phase to the implementation it takes about 3.27 ms. This is the median of the collected runtimes. See figure 4.6 for the box chart.

The rest of the center circle processing takes another 240 ms. However using the remaining $\mathcal{D}$-Phase bits to store if the pixel was already visited cuts the runtime down by 15 ms. But despite the optimizations and claims from the paper it was not possible to get a running real-time implementation for finding ellipses. See figure 4.7 for a box plot comparison of the non optimized and optimized parts.

Also notice that the optimization introduces a lot of variance which is equally bad for a realtime application.

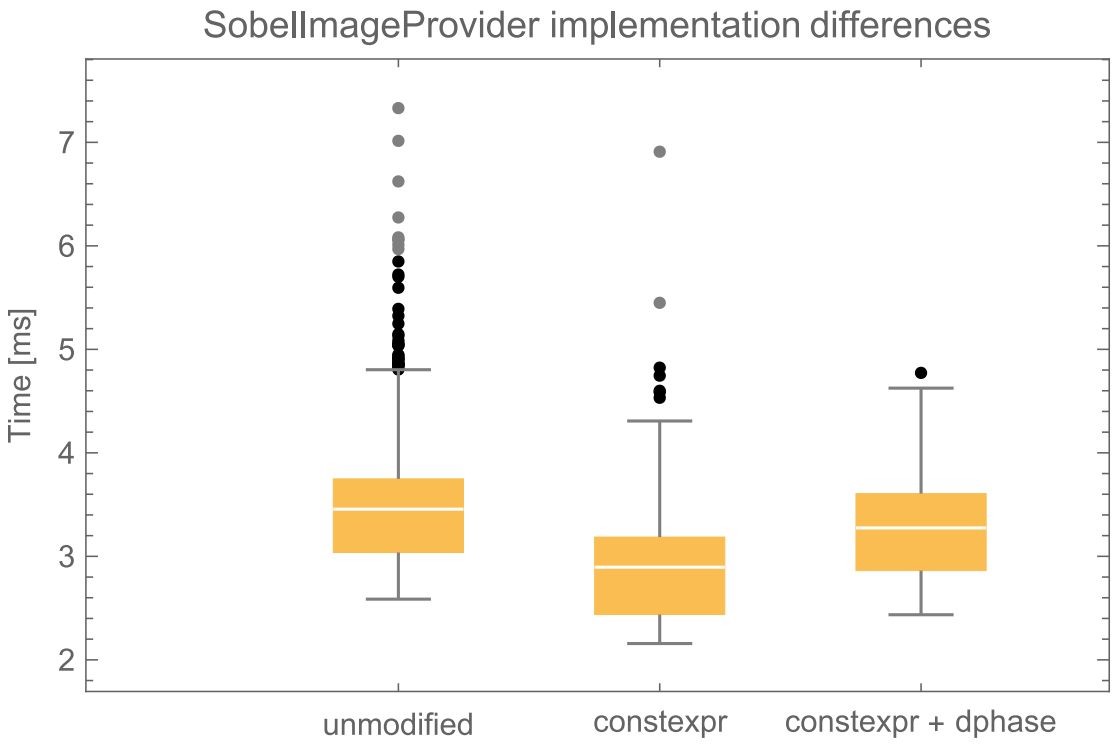


Figure 4.6: *Comparison of the different stages of the SSE Sobel implementation.*

4.4 Problems of chosen method

In the process of implementing and testing there were several problems found that would be crucial for better fitting performance.

Sobel instead of canny Because the canny edge detection was changed to the sobel edge detection it happens sometimes that arcs from the inner and outer boundary are collected into one arc. This leads to bad center circle estimates because points from the inner arc are used to find a parallel chord together with points from the outer arc.

Modified $\mathcal{D}$-phase Also due to the modified implementation of the $\mathcal{D}$-phase the detection on some ellipses were better but on others worse.

Not rejecting the pixels on the decision boundary leads to longer arcs. This is generally better for the estimation and fitting. But it also leads to much more overlapping arcs. These will not be matched together in the arc selection phase because then it is not true anymore that arcs from different parts of an ellipse can matched together by their position.

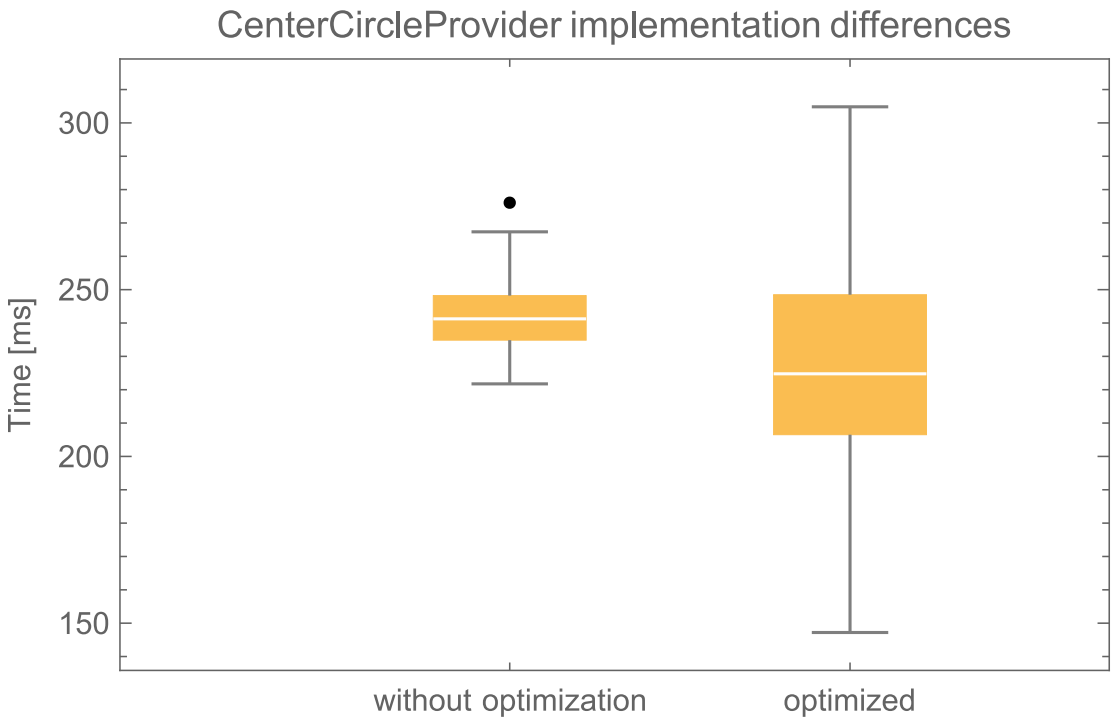


Figure 4.7: *Comparison of the center circle detection with and without storing the visited data in the image memory.*

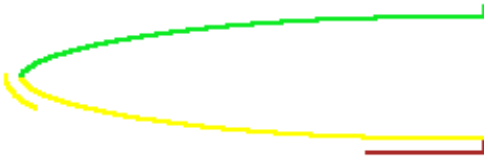
Another problem lies within the current method used to calculate the parallel chords. With heavily overlapping inner and outer arcs the parallel chords are sometimes calculated within the overlapping area which produces very bad center estimates.

Center circle estimation The center circle estimation step has problems with robustly finding the correct center. Especially when the width to height ratio of the ellipse increases bigger the center is further away from the parallel chords so the accuracy of the estimate gets much worse. This stage has multiple problems.

If the center is further away the numerical accuracy gets much more important and can lead to problems with floating point precision. Errors in not perfectly parallel lines makes the estimate much less accurate as well. Also the limited resolution of the pixels itself could be the problem for the bad estimates.

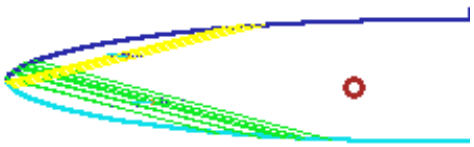
An example of a very bad center circle estimate can be seen in figure 4.8 the algorithm was able to correctly estimate the center in one pairing but in no other one. This way it could not proceed to the next stage.

(a) *Original image.*

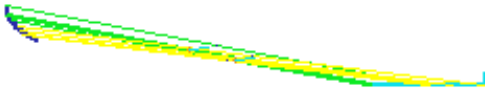


(b) *Extracted and classified arcs.*

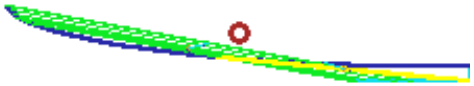
(c) *First arc matching with parallel chords and center estimate.*



(d) *Second arc matching with parallel chords and center estimate.*



(e) *Third arc matching with parallel chords and center estimate.*



(f) *Fourth arc matching with parallel chords and center estimate.*

Figure 4.8: *The different stages of the ellipse fitting for a bad example. Only a few arcs gets extracted and only a few arcs get paired up. Also the center estimates are quite bad. The algorithm was not able to match triples together.*

Partially visible ellipses Another problem is that the algorithm needs to see three out of the four quadrants of an ellipse to be able to robustly calculate the parameters. On a real soccer field there will be other robots possibly occluding the circle or the robot's head is turned slightly away from the center circle.

Runtime The method is not fast enough for real-time usage on the NAO platform which renders it unusable in the current situation. To get the method into a feasible runtime region the algorithm needs to be sped up at least 10 times.

Fitting conics It would be good if it was also possible to not only fit an ellipse but the more general conic to be able to detect the center circle while inside it.

Chapter 5

Conclusion and Outlook

Conclusion It was not possible to meet the goals stated in 1.5 with the chosen methods. After implementing and optimizing the fast and effective ellipse detection method it was not possible to run it in realtime on the NAO.

Despite the initial suspicion that the sobel/canny edge detection could be the slowest part of the algorithm this was not the case. All other parts of the fast and efficient ellipse detection were far slower than the edge detection part. Even up to the point that it is not feasible to run it in realtime on the robot.

Beside that, the chosen method showed other problems within the different stages that could only partially be solved. It was possible to reduce some of the larger problem for corner cases that one of the center line's slope tends to infinity. But it was not possible to further reduce the errors on the accuracy of the estimates.

It could be shown that it is feasible to compute the sobel edge detection for the whole image in under 3 ms. It is likely that this module will be included in the current source code of the framework so that other people are able to use it for further development of a sobel based vision pipeline.

Outlook The league will try to expose the games to more external lighting conditions such that it will be much harder to use the current methods for feature extraction. More research needs to be done in the direction of 2d edge-detection-based feature extraction to achieve higher reliability against varying lighting conditions. This thesis is a good start for this goal, but showed that standard methods are too slow for the current hardware iteration.

As mentioned in the introduction, a new NAO version with a more powerful CPU will be released in this year. It is possible that the method will be feasible on the new hardware.

This thesis showed another point that needs more research. While the robot is inside the center circle the center circle can not be fitted with an ellipse. A more general cone fitting method is needed here.

Bibliography

[Alb+16] Christian Albrecht et al. *Final Report of Project Group 590*. Tech. rep. Technical University of Dortmund Robotics Research Institute, 2016.

[Bra00] G. Bradski. „The OpenCV Library“. In: *Dr. Dobb's Journal of Software Tools* (2000).

[Bro+12] Ilja N Bronstein et al. *Taschenbuch der Mathematik*. Springer-Verlag, 2012.

[FPC14] Michele Fornaciari, Andrea Prati, and Rita Cucchiara. „A fast and effective ellipse detector for embedded vision applications“. In: *Pattern Recognition* 47.11 (2014), pp. 3693–3708.

[Gei14] David Geier. *Implementation of the rotating calipers method in JavaScript*. https://github.com/geidav/ombb-rotating-calipers. Last visited: 2018-04-27. 2014.

[HUL17] HULKs. *HULKs Code Release*. https://github.com/HULKs/HULKsCodeRelease/tree/coderelease2017. Last visited: 2018-05-23. 2017.

[Int17] Intel. *Intel VTune Amplifier*. https://software.intel.com/en-us/intel-vtune-amplifier-xe. Last visited: 2017-05-23. 2017.

[Kit+97] Hiroaki Kitano et al. „Robocup: The robot world cup initiative“. In: *Proceedings of the first international conference on Autonomous agents*. ACM. 1997, pp. 340–347.

[Kog13] Linor Kogan. *Circular Values Math and Statistics with C++11*. https://www.codeproject.com/Articles/190833/Circular-Values-Math-and-Statistics-with-Cplusplus. Apr. 2013.

[Mat91] Jii Matouek. „Randomized optimal algorithm for slope selection“. In: *Information processing letters* 39.4 (1991), pp. 183–187.

[Nao17] Nao-Team HTWK Leipzig. *HTWKVision Code Release*. https://github.com/NaoHTWK/HTWKVision. Last visited: 2018-05-05. 2017.

[PL10] Dilip K Prasad and Maylor KH Leung. „Clustering of ellipses based on their distinctiveness: An aid to ellipse detection algorithms“. In: *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*. Vol. 8. IEEE. 2010, pp. 292–297.

[PLQ13a] Dilip K Prasad, Maylor KH Leung, and Chai Quek. „Ellifit: an unconstrained, non-iterative, least squares based geometric ellipse fitting method“. In: *Pattern Recognition* 46.5 (2013), pp. 1449–1465.

[PLQ13b] Dilip K Prasad, Maylor KH Leung, and Chai Quek. *Matlab implementation of Ellifit*. https://sites.google.com/site/dilipprasad/Source-codes. Last visited: 2018-04-23. 2013.

[Qt 17] Qt Company. *Qt Framework, Version 5.9*. Oslo, NO. 2017.

[Röf+17] Thomas Röfer et al. *B-Human Team Report and Code Release 2017*. http://www.b-human.de/downloads/publications/2017/coderelease2017.pdf. Last visited: 2018-05-05. 2017.

[Sof18] Softbank Robotics Corp. *NAO Robot V5 Video Camera Specifications*. http://doc.aldebaran.com/2-5/family/robots/video_robot.html. Last visited: 2018-04-18. 2018.

[Tou83] Godfried T Toussaint. „Solving geometric problems with the rotating calipers“. In: *Proc. IEEE Melecon*. Vol. 83. 1983, A10.

[WH08] Lili Ayu Wulandhari and Habibolah Haron. „The evolution and trend of chain code scheme“. In: *Graphics, Vision and Image Processing* 8.3 (2008), pp. 17–23.

[Wik18] WikiBooks. *Convex hull/monotonic chain*. https://en.wikibooks.org/wiki/Algorithm_Implementation/Geometry/Convex_hull/Monotone_chain#C++. Last visited: 2018-04-27. 2018.

[Wol17] Wolfram Research, Inc. *Mathematica, Version 11.2*. Champaign, IL. 2017.

[YC94] Peng-Yeng Yin and Ling-Hwei Chen. „New method for ellipse detection by means of symmetry“. In: *Journal of Electronic Imaging* 3.1 (1994), pp. 20–30.