TANSZÉKVEZETŐ

DIPLOMATERVEZÉSI FELADAT

Danyi Dávid
Villamosmérnök hallgató részére

Marker alapú helymeghatározás képfeldolgozással

Az információfeldolgozási kapacitás nagymértékű növekedésével párhuzamosan egyre szélesebb körben váltak alkalmazhatóvá (valós időben) képfeldolgozási, gépi látás alapú eljárások. Az ideálistól eltérően a valós képek feldolgozását számos hatás nehezíti, úgy mint zaj, optikai torzítások, megvilágítás- és színbeli különbségek, stb.

A feladat témája aktuális, az egyre gyakrabban alkalmazott mobil robotikai, kiterjesztett és virtuális valóság alkalmazásoknál minden esetben felmerül a „hol vagyok?" kérdés, azaz a nézponti paraméterek becslése, lokalizáció. A SLAM (Simultaneous Localization and Mapping) algoritmus számos szenzorforrás által biztosított információval alkalmazható, így kamerával is. Az általános megoldás tájékozódási pontok egymáshoz képesti elhelyezkedését becsli, a mérési bizonytalanságot folyamatosan csökkentve, valamint méri a tájékozódási pontokhoz képest a megfigyelő pozícióját.

Jelen munka is a fenti témához, kapcsolódik, cél megvizsgálni egy olyan mesterséges, passzív marker megvalósíthatóságát, ami bizonyos paraméterekben (azonosíthatósági tartomány, részleges láthatóság) jobbat kíván biztosítani, mint az elterjedt (ARtag, glyph, stb.) megoldások. A marker egy oldalukon nyitott különböző méretű és alakú négyszögeket tartalmaz. A diplomaterv célja megvizsgálni a javasolt marker jellemzőit és használhatóságát.

A hallgató feladatának a következőkre kell kiterjednie:

- Mutasson be pozícióbecslő algoritmusokat, röviden ismertesse ezek működését!
- Hasonlítson össze különböző nézőpontbecslési algoritmust síkban elhelyezkedő pontpárok alapján!
- Készítsen egy marker felismerő megoldást!
- Végezzen méréseket (ideális és valós képeken) a marker által meghatározott pozíció pontosságára vonatkozóan!
- Hasonlítsa össze és értékelje az eredményeket!

Tanszéki konzulens: Kovács Viktor, tanársegéd

Budapest, 2017. február 18.

Dr. Charaf Hassan
egyetemi tanár
tanszékvezető

Budapest Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

H-1117 Budapest, Magyar tudósok krt 2. Q.B.207
Telefon: 463-2870. Fax: 463-2871
www.aut.bme.hu

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

Marker Based Localisation and Pose Estimation Using Image Processing

Thesis

Author
Dávid Danyi

Consultant
Viktor Kovács

May 2, 2018

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Dávid Danyi*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelmûen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerzõ(k), cím, angol és magyar nyelvû tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhetõ elektronikus formában, a munka teljes szövegét pedig az egyetem belsõ hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetõvé.

Budapest, May 2, 2018

Dávid Danyi
hallgató

Kivonat

A képfeldolgozás nem újkeletű tudományterület, már évtizedek óta folynak kutatások ezen a téren. Sok, ma is használt algoritmust az 1960-as években fejlesztettek ki. Akkoriban főleg a tudósok körében használt, drága eszköznek számított a számítógépes képfeldolgozás. Műholdképek, orvosdiagnosztikai adatok elemzésére, optikai karakterfelismerésre használták. Az olcsó és nagy teljesítményű, általános felhasználású számítógépek terjedésével azonban új lehetőségek nyíltak meg ezen a területen. Lehetővé vált például a valósidejű képfeldolgozó algoritmusok futtatása. Ezen fejlődés nélkül lehetetlen lett volna a 3 dimenziós látórendszerek kifejlesztése. Ezek a rendszerek jóval számításigényesebbek a klasszikus képfeldolgozási problémáknál, de a mai technológiával már ezek a megoldások és elérhetőek az átlagos felhasználók számára. Ennek legfőbb jele a robot navigációs, virtuális- és kiterjesztett valóság alkalmazások széleskörű megjelenése.

Jelen munka a fiduciális markerek alapján történő nézőpont meghatározás alkalmazhatóságát vizsgálja. A nézőpont-meghatározás célja a kamera pozíciójának és orientációjának meghatározása egy ismert markerhez viszonyítva. Ez egy összetett feladat aminek a megoldása több képfeldolgozási- és optimalizációs feladat megoldását igényli. Ez a dolgozat be fogja mutatni a kép alapján történő nézőpont-meghatározás lépéseit.

A munka első részében ismertetésre kerül a nézőpont-meghatározási probléma. Először röviden össze lesz foglalva a P-n-P néven ismert probléma: a nézőpont meghatározás n pontpár alapján, aminek ismert a világkoordinátákban adott helyzete, valamint a képi helyzetük is. Ezt a problémát többen többféleképp megoldották, néhány ilyen megoldás alapvető gondolatmenete összefoglalásra kerül. Ennek a szakasznak a zárásaként összehasonlítom az eljárásokat és kiválasztom a tulajdonságaik alapján a projekt céljára a legideálisabbat.

A pozicionálás pontosságát és robusztusságát nagyban befolyásolhatja az alkalmazott fiduciális marker is. Vannak ugyan már elterjedt markertípusok (ARTag, glyph, stb...), de jelen munkában kísérletet teszek egy új marker tervezésére. Ez az új marker megpróbál jobb eredményt nyújtani bizonyos területek, mint a már elterjedt megoldások. Ezen marker alkalmazhatósága is vizsgálva lesz ebben a dolgozatban.

A dolgozat utolsó nagyobb része a markerek felismerésére használt képfeldolgozási eljárásokról fog szólni. A különböző algoritmusok rövid elméleti áttekintése után egy-egy implementációs javaslat is közlésre kerül. A felismerő eljárások hatékonysága is vizsgálat alá

kerül, ideális és zajos képeken egyaránt. A mérési eredmények alapján javasolni fogok egy, a projekt számára optimális markerfelismerő eljárást.

Abstract

Image processing has been an intensively researched subject for decades. Many algorithms that are used today have been developed in the 1960s. At that time, it was a costly tool mainly used by scientists for satellite imagery, medical imaging, optical character recognition, etc... The advancement of cheap and powerful general purpose computers opened up new possibilities for research and application. Real time image processing became possible. An interesting and even more computationally expensive sub-field of computer vision is 3D reconstruction. With today's (consumer) technology it is possible to map the 3D world based on image processing solutions. Navigational, Augmented and Virtual Reality applications are spreading.

This paper will examine the use of fiducial markers for camera pose estimation using a single camera. The goal of pose estimation is to determine the position and orientation of the camera with respect to a known marker. This is a complex task, which involves multiple image processing steps, as well as solving optimization problems. This work will provide an overview of the steps necessary for estimating the camera pose based on picture of a fiducial marker.

A section of this work will be dedicated to the pose estimation problem. There will be a short summary of the problem of reconstructing the view point based on point pairs in the world coordinate system and image points. Then some algorithms will be summarised that solved that problem. This section will be closed by comparing the benefits and drawbacks of these algorithms and choosing the one that best suits the need of this project.

The choice of the marker also influences accuracy and robustness of the pose estimation solution. There are already some marker types available for use (ARTag, glyph, etc...). This paper also proposes a new marker type which tries to offer better performance than the aforementioned solutions. The applicability of the new marker will be examined in various conditions.

The last major part of this work is about the different possible methods for extracting the markers from the images. In that section there will be short theoretical summaries of the detection methods. After the theory is covered, implementations of the aforementioned methods will be recommended. The performance of the detection algorithms will also be benchmarked on optimal and noisy images. Based on the tests results an optimal method will be selected.

Abbreviations

This is a complete list of the abbreviations used in this paper.

DOF Degrees of freedom

RQIM Random Quad Image Marker

SHT Simple Hough Transformation

RHT Randomised Hough Transformation

PPHT Progressive Probabilistic Hough Transform

LSD Line Segment Detector

LLA Level-Line Angle

GWN Gaussian White Noise

Introduction

Computer vision, and image processing in general, is a computationally intensive area. In the past the use of these algorithms was severely limited by the lack of processing power. Image processing solutions were mostly used for scientific purposes, and the algorithms ran offline: real-time applications were not possible. Satellite photos were analysed, medical imaging solutions were developed at the time. Optical character recognition was also a popular topic for image processing research. A famous scientific example from that time gave the basis for the Hough transformation, which will also be discussed in this work. The transformation was developed to automatically analyse bubble chamber photographs.

With the developing technology, specifically semiconductor manufacturing, more and more possible uses for image processing began to appear. Around the 1970s cheaper computers and dedicated hardware solutions started spreading. This made it possible to create real time image processing applications for some use-cases. One such use-case was television standards conversion.

As general purpose computers became faster and cheaper, they replaced the specialised circuits in almost all areas of application. Nowadays image processing solution to common problems (localisation, mapping, measurement, etc...) is chosen as a solution because it became the cheapest and most versatile alternative. Furthermore, 3D computer vision applications became not only possible, but widespread. 3D scanners, range finders, virtual- and augmented reality solutions have spread from laboratories and research institutions to consumer electronics. Processing power is no longer a bottleneck for most computer vision applications.

Chapter 1

Pose estimation

1.1 Pose Estimation Algorithms

1.1.1 Fast and Globally convergent Pose Estimation

1.1.2 Linear Pose Estimation from Points or Lines

1.1.3 Robust Pose Estimation from a Planar Target

1.2 Comparison

Chapter 2

Markers

One of the goals of this project was to design a fiducial marker with advantageous properties for use in pose estimation. In a typical scenario the marker may be seen from largely varying viewpoints, therefore it has to have some level of scale invariability. If the observer is far from the marker, the smaller details may be lost due to the limited resolution of the camera. If the same observer moves closer to the marker, it may fill the whole field of view and some features may even slip off the image. This leads to another feature the marker needs to have: redundancy. If the observer gets too close to the marker or some obstacle partially blocks the view, the localisation still needs to provide usable results.

The intended use of the markers is spatial localisation and pose estimation. In other words: approximating the observers 3D coordinates (x, y, z) and orientation (ϕ, θ, ψ) with respect to the marker. It is supposed that the observer uses a single camera system for navigation (e.g. smartphone or robotic application with limited resources). This means the marker needs at least 6 degree of freedom.

To sum up the above discussed specifications, a suitable marker would have to:

- have at least 6 DOF
- be (to some degree) scale invariant
- have redundancy

In the following sections will be a recommendation for a marker conforming for the listed specifications. It is based on 3 connected line segments forming a quad with one missing side. The whole marker is built from quads with different side lengths and angles.

2.1 Quad

A marker is put together from quads. Figure 2.1. shows two examples. One side of the quads is left out: they are put together from three joint line segments. The middle segment, with

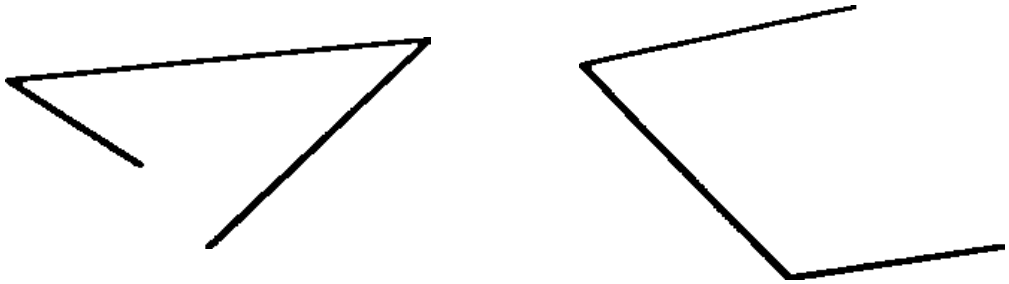

Figure 2.1: *Example for different quads*

two adjoining lines, will be referred to as the 'base' of the quad. The outer segments are going to be called 'arms'.

A quad has 6 degrees of freedom. There are 3 independent distance parameters: the length of the base and the two arm segments. There are also 3 unrelated angle parameters: the angles between each arm and the base, and the orientation of the quad.

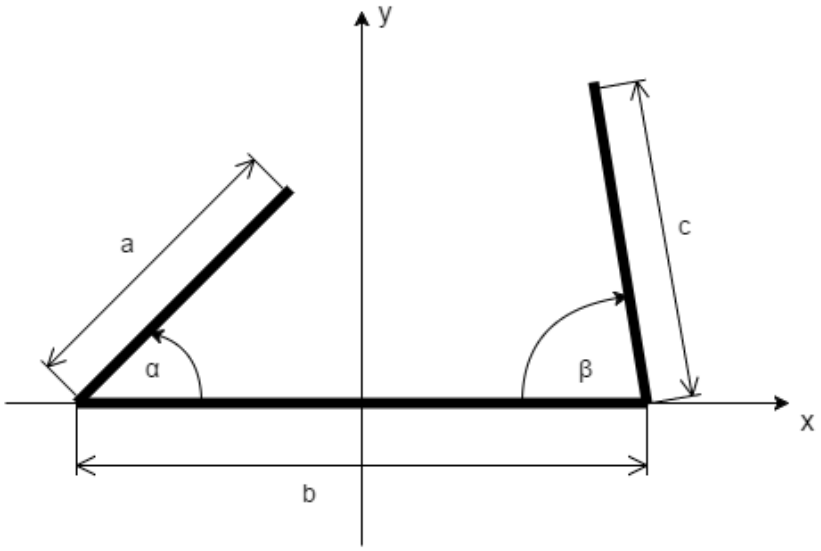

Figure 2.2: *Quad parameters*

Figure 2.2. shows the free parameters of a quad (the orientation is not shown on the image). The following notation is used:

a : The length of one arm

b : The length of the base

c : The length of the other arm

α : The angle between one arm and the base

β : The angle between the other arm and the base

γ : The angle with which the whole quad is rotated

For the sake of simplicity, figure 2.2. does not show the rotation with γ. The quad would be rotated around the origin of it's coordinate system.

The values of the length parameters are given in pixels, although they can be expressed in any unit of distance. The angles can be given is degrees or radians (in the implementation degrees are used for easier human readability).

$$a \in (0, a_{max}] \tag{2.1}$$

$$b \in (0, b_{max}] \tag{2.2}$$

$$c \in (0, c_{max}] \tag{2.3}$$

$$\alpha \in (0, 180^\circ) \tag{2.4}$$

$$\beta \in (0, 180^\circ) \tag{2.5}$$

$$\gamma \in [0, 360^\circ) \tag{2.6}$$

Equations (2.1) through (2.6) specify the range of each parameter. The maximum of the distance parameters are set by the space left on the image for the given marker, there is no theoretical limit for them. There is also no constraint for the resolution of the parameters. From the applications point of view, there are quads with continuous[1] and discrete parameter spaces.

2.1.1 Quad representation

There are several ways to represent quads, each with different advantageous properties. For this work multiple considerations were made in that regard. The most straightforward is to simply store the above mentioned parameters. This is simple and easy for human reading, which is great help in the development process.

A step forward from this is to norm the a, b and c parameter of the quad with the base segment's length. Then the following parameters are used:

s : marker size, the same as the base length

m_a : 'a' multiplier. $m_a = a/b$

m_c : 'c' multiplier. $m_c = c/b$

The α, β, γ angle parameters are not changed. This gives a scale or size parameter for the quad, which is useful for marker generation. These two representations are good for development and marker generation, but not so much for calculations.

[1] That is, only limited by the computational precision

A third option is to store the endpoints of the line segments. It requires the storage of 4 points: two endpoints (E_1, E_2) and two inner points (I_1, I_2). Equations (2.7) through (2.10) define the points' coordinates before rotating with γ, using figure 2.2.'s notation.

$$I_1' = (-\frac{b}{2}, 0) \tag{2.7}$$

$$I_2' = (\frac{b}{2}, 0) \tag{2.8}$$

$$E_1' = (-\frac{b}{2} + b * cos(\alpha), a * sin(\alpha)) \tag{2.9}$$

$$E_2' = (\frac{b}{2} - c * cos(\beta), c * sin(\beta)) \tag{2.10}$$

$$Rot(\gamma) = \begin{pmatrix} cos(\gamma) & -sin(\gamma) \\ sin(\gamma) & cos(\gamma) \end{pmatrix} \tag{2.11}$$

The point E_1, E_2, I_1, I_2 can be obtained from E_1', E_2', I_1', I_2' by a multiplication with the rotational matrix $Rot(\gamma)$.

That method is redundant for storage: it uses 8 parameters instead of 6. However this poses no practical problem in the scope of the project. The one outstanding benefit of this method is it's efficiency in calculations. Because it is based on points in a Euclidean space, linear algebraic methods (matrix multiplications) can be used for calculating the projective transformations.

In this project the second and the third options are used. The first, naive method is omitted because it has no considerable advantage over the other two. The second method, using the size, multiplier and angle parameters is used in marker generation. The third is used during the calculations and the recognition process.

2.2 Marker

Quads are 6 DOF shapes: in theory it would be enough to use only one of them for localisation and pose estimation. However that method would have very low error tolerance and questionable accuracy even in a best case scenario. To comply with the specifications written in the beginning of this chapter, the markers are put together from multiple quads. By placing quads with different orientations and sizes the error tolerance and accuracy can be greatly improved.

An intrinsic positive quality of using multiple quads with varying sizes is the scale invariance. As mentioned, even a single quad is sufficient for the task at hand. If the smaller quads become unrecognisable because of the low resolution or too large distance, a successful measurement is still possible. The same is true on the other end of the spectrum: if the observer is too close to the marker and the larger ones leave the field of view, the position and orientation can be calculated from the smaller quads.

Figure 2.3. shows an example for a marker. It is generated with the simple algorithm described in the next section, and is not optimal in many ways. Nonetheless it is functional, even if only a fraction of the quads are registered for the measurement.

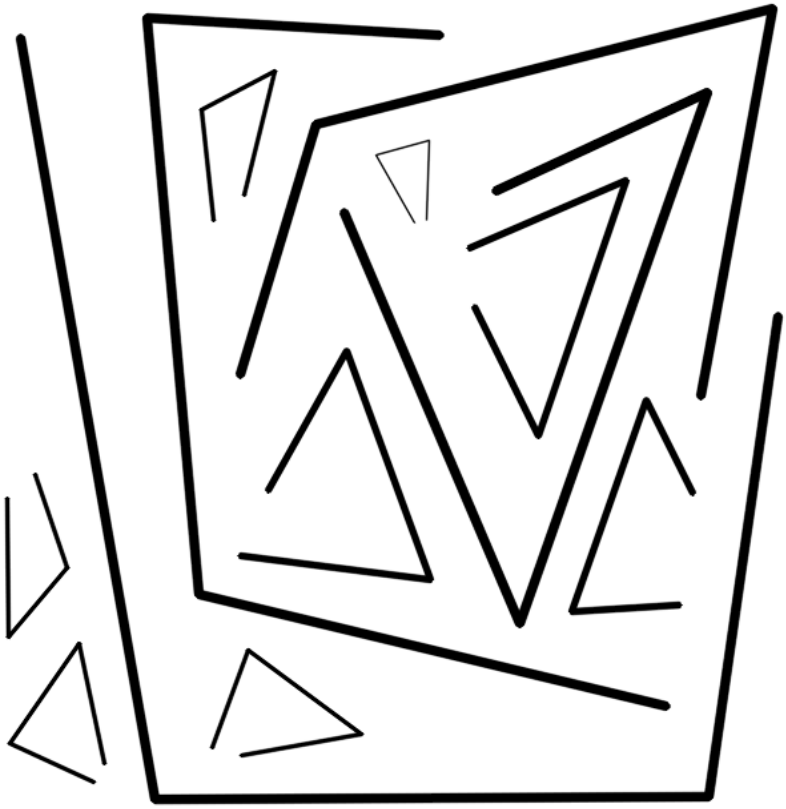

Figure 2.3: *An example for a marker*

The markers are going to be referred to as RQIM, which means Random Quad Image Marker. As the name suggests, the quads are randomly generated and placed on the markers.

Unless otherwise specified, RQIMs use quads with continuous parameter spaces. In this chapter there will also be a small introduction to discrete parameter markers and their potential applications.

2.2.1 Marker generation

In the current state of the project, markers are randomly generated using a simple algorithm. The generator routine receives the number of quads to be used in the current RQIM. The core concept is to create the desired number of random quads and place them on the image.

Let the number of quads to generate be n. First, the quad sizes are picked. There is an upper and a lower limit for them, given in percent of the image size. The generated sizes

follow an exponential distribution:

$$s = e^{-x*f} \tag{2.12}$$

Where s is the quad size, x is random number between 0 and 1 with uniform distribution, and f is a scale factor. Then the n sizes are ordered in descending order.

After the scale factors are picked, the whole quads are generated by the following method. A random quad is created with the first (the largest) scale and placed on the image. Then another quad is created with the next largest size. After every new quad a check is performed whether or not it can be placed on the marker. If it cannot, then a new quad is generated with the same scale factor until it can be placed or the algorithm reached the limit of retries.

With this simple logic n quads are placed on the RQIM and the creation process is finished. Below is the pseudo-code of the algorithm.

```
n_max = number of quads to create
f = scale factor for exponential distribution
lowlim = lower size limit
uplim = upper size limit
n = 0
while n < n_max
        size = exp(-rand() * f)
        if size > lowlim and size < uplim
                store size
                n = n+1
        endif
end
sort(sizes, descending)
n = 0
while n < n_max
        while quad placed or max tries
                quad = create_random_quad(sizes(n))
                if quad can be placed
                        place quad
                        n = n+1
                end
        end
end
return marker
```

This method is not optimal and is based on trial and error, but it gives usable markers for the development process.

2.2.2 Discrete RQIM

There are experiments in progress with discrete parameter space quads. It may be advantageous to quantize the parameter space in order to decrease the error probability in the pose estimation process.

Quads with finite possible states can be stored using much less resources than their continuous counterpart. As an example let us take a look at the following quantisation.

Angles: $15°, 30°, 45°, 60°, 75°, 90°, 105°, 120°$

Multipliers: $0.40, 0.60, 0.80, 1.0, 1.25, 1.50, 1.75, 2.0$

Orientations: $0°, 22.5°, 45°, 67.5° \ldots 270°, 292.5°, 315°, 337.5°$

Sizes: $1, 0.8, 0.6, 0.5, 0.4, 0.3, 0.25, 0.2, 0.1, 0.08, 0.06, 0.05, 0.04, 0.03, 0.025, 0.02$

In this example, there are 8 possible values for the angle parameters, also 8 for the multipliers, 16 for the orientation and also 16 for the sizes. If the possibilities are stored in a lookup table, it is enough for the quad to store the index at which the value is accessible. A quad is defined by two angle parameters, two multipliers, an orientation and a size. The angles (in this case) require at least 3 bits each, the multipliers also. The orientation and the size need 4 bits each. That gives a sum of 20 bits per quad, which is significantly less than the space required to store 6 floating point numbers per quad.

This 20 bit word is also usable as an ID for the quad. It may be possible to code information in these ID-s, so the marker could provide additional information. This information could be related to and used by the localisation process, or be totally unrelated, general data. These possibilities have not yet been extensively researched.

The discrete RQIMs are usually less dense than the continuous ones, due to the limited angle possibilities. This means fewer quads per marker, which leads to decreasing redundancy. An optimum must be found between the number of quads per RQIM and distance between quads in the parameter space.

Chapter 3

Quad detection

In this chapter there will be a summary of the image processing algorithms tried and used for the recognition of the fiducial markers. The input of this recognition step is the image taken by the camera, and the output is a list of quads belonging to the marker visible on the image. As a common preprocessing step for all quad detection methods segmentation is performed on the input image: quad-like blobs are extracted and separately passed to the quad detector logic. This step of the *marker recognition process* takes the segmented input image and initialises quad structures based on the observed picture. The quad structures are then passed to the next processing step: the pose estimation logic.

The process here diverges depending on which quad detection algorithm is used. They all need differently conditioned input images for optimal performance. From a computer vision point of view the task is to detect joint line segments. This is a well researched task in image processing, there are many well tried algorithms for it. For example, the problem can be solved by detecting lines and finding their intersection, or detecting corners and figuring out how they are connected, etc... The detection routines not necessarily have the same output format[1], so conversion may also be needed.

Three separate quad detection techniques and their variants were profiled in this experiment.

- Hough-transformation
- Corner detection
- Line Segment Detector[4]

The first one uses the Hough-transformation for line detection. There are many variants of the transformation: Standard Hough Transform, Probabilistic Hough Transform, Multiscale Hough Transform, etc... The 2 most commonly used are the standard- and the

[1]Some return line segments defined by their endpoint, others use the polar representation of a line etc.

probabilistic variants. The OpenCV framework offers implementations for them, both were tested in the experiment.

The second detector is based on corner recognition. There are more variants of this method to try out, too. The corner metrics of a feature can be calculated differently with (Harris metric, eigenvalues, etc.) varying results. It is also needed for the solution to be scale invariant, which also can be achieved in a number of ways.

The third alternative is the Line Segment Detector algorithm described in [4]. It is a robust and fast algorithm for detecting line segments on an image. The OpenCV framework provides an implementation of it as well.

A typical marker shot with partial visibility is shown figure 3.1.. In this chapter will be a

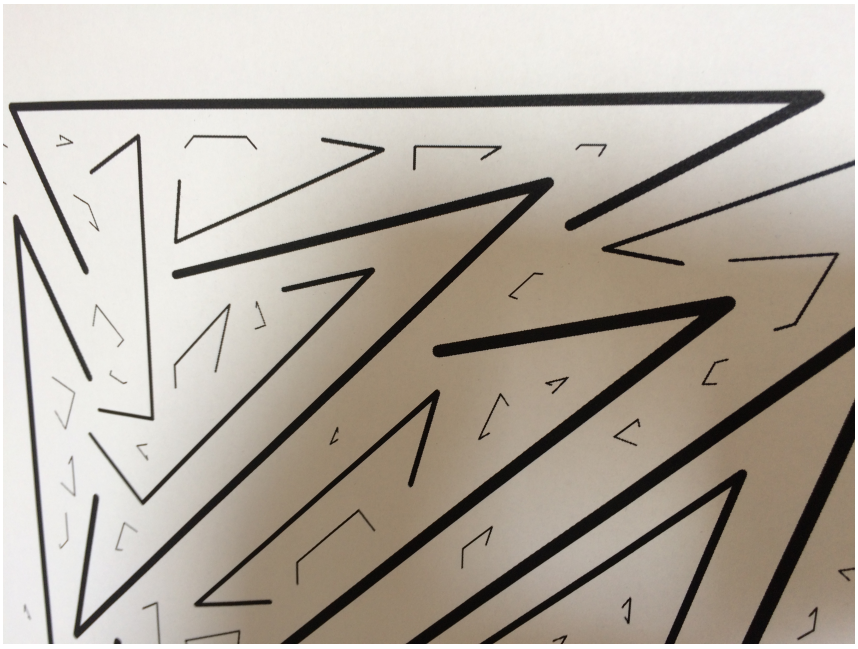

Figure 3.1: *Partially visible marker (taken with commercial smartphone)*

short summary of the algorithms used for testing and performance comparison.

3.1 Theoretical Overview

Before going over how the image processing algorithms were applied to achieve quad detection, a short theoretical overview of the used algorithms will be presented. To solve the problem at hand (i.e. to detect quad instances on an image) multiple well known image processing algorithms were used. For line detection two variants of the Hough transform (Standard[3] and Probabilistic[8]) and a fundamentally different algorithm, the **LSD** was used. As mentioned before, not only solutions based on line detection were tried during the course of this work; corner detection methods were also tried. The Harris corner detector[5] and it's improved version the Shi-Tomasi detector[9] were compared.

Although the implementation of the aforementioned algorithms were provided by the OpenCV framework, it was far from unnecessary to understand how each algorithm works. They show their optimal performance on differently conditioned inputs. For example, corner detection works well on "raw" images, while the Hough-transform based solutions need edge images of skeletons to perform. It was important to know the limitations of each solution. All in all, the understanding of the inner workings of the algorithms used was helpful in choosing the "right tool for the job".

In this section there will be the theoretical overview of the above mentioned algorithms, with some historical context. Their comparative advantages for this project will also be highlighted.

3.1.1 Hough transformation

One of the most commonly used methods for line detection on images is the Hough transform. Over it's long history many publications have been made about it's applications, performance and improvements.

Originally it was developed by Paul Hough in 1959 and later patented in 1962[6]. It was intended to be used for machine analysis of bubble chamber photographs. In it's modern form (with the $\theta - \rho$ parametrisation) was introduced in 1972 by Duda and Hart[3]. The transformation became popular in the image processing community after Ballard's article[1] about generalising the algorithm for detection of arbitrary shapes. There were many optimised and improved variants of the transformation, however the basic concept remained the same. In 1990 a publication[10] introduced the Randomized Hough Transform, which was a fundamentally new approach to the algorithm with notable merits. As opposed to the one-to-many mapping of the simple Hough transform, the randomised version uses a convergent many-to-one mapping when creating the parameter space.

In this work the Standard Hough Transform and one of it's optimised versions, the Progressive Probabilistic Hough Transform will be used. The PPHT, although being probabilistic, doesn't belong to the class of randomised Hough transforms. It uses the same one-to-many mapping as the SHT. The OpenCV framework provides implementations for the SHT and the PPHT, which is one of the main reason why they were chosen for this project.

After this short historical overview the theory of the transformations will be discussed.

Standard Hough Transform

The transformation is used to find instances of a model on digital images. The models are usually simple geometric shapes like lines, circles or ellipses. The curves are described by their parameters, e.g. slope and intercept for a line, centre point and radius for a circle etc.. Every non-zero pixel[2] votes for the features it could be part of. The number of votes is

[2]The transformation works on binary images

stored for every possible parameter combination. Then a threshold is applied to the stored votes, and the remaining parameters are accepted as model instances.

At first Hough described the algorithm to lines, but later the method would be generalised to any analytic[3] curve or shape. This theoretical overview is based on the example of line detection. The process is the same for every analytic curve, the only difference is the parameter space's dimension. The original patent[6] used the slope-intercept representation of lines.

$$y = m * x + b \tag{3.1}$$

In this case, the *parameter space* is 2 dimensional and it's axes are m and b. Every point in the parameter space represent an image space line. With this representation every non-zero pixel in the image space transforms into a line in the parameter space. For a given (x_0, y_0) pair (3.2) gives the line in the parameter space.

$$b = -x_0 * m + y_0 \tag{3.2}$$

Collinear points in the image show up in the parameter space as intersecting lines. The more lines intersect in a given (m_0, b_0), the more likely it is the image contains the $y = m_0 * x + b_0$ line. The problem with this parametrisation is that the parameter space is unbounded along both axes. Both intersect and slope can have values in the range of $(-\infty, \infty)$. Duda and Hart[3] proposed an alternative parametrisation, which turned out to be better for application. They used the *normal parametrisation* of a line, shown in (3.3).

$$\rho = x * cos(\theta) + y * sin(\theta) \tag{3.3}$$

In (3.3) ρ means the distance of the line from the image plane's origin. θ is angle of the normal vector of the line. If the *normal parametrisation* is used the parameter space

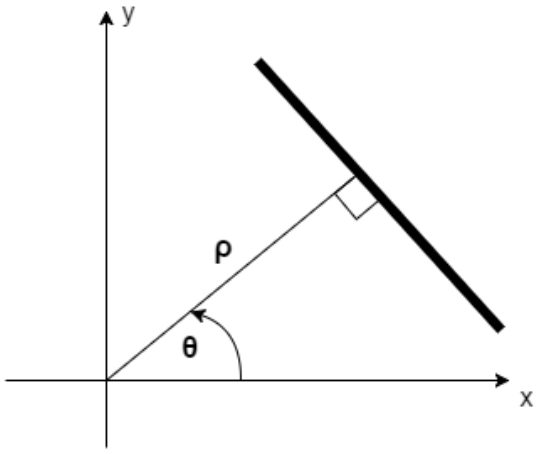

Figure 3.2: *Normal line parameters*

becomes finite in both dimensions. θ is in the range of $(0, 2\pi)$, ρ is bounded by the image

[3]The Generalised Hough Transform even extends to arbitrary shapes

size. In this case the image points define sinusoid curves in the parameter plane, and the line detection is done by searching for their intersections.

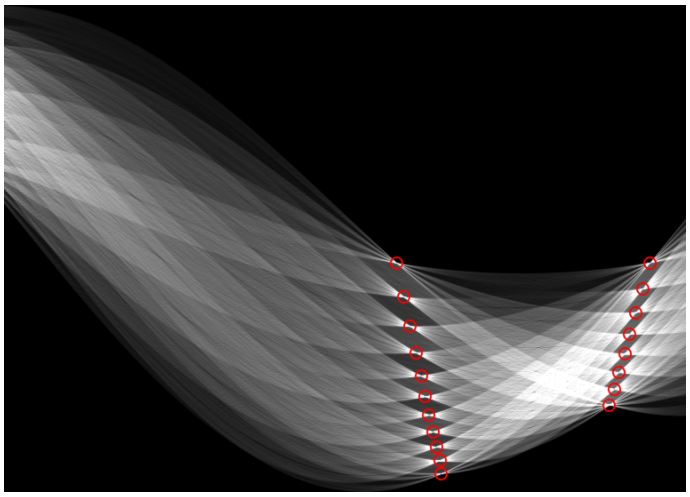

Figure 3.3: *Hough-transform of a chessboard pattern*

As mentioned before, the line detection is based on a voting scheme. The parameter space (in this case a 2 dimensional plane) is divided into *bins*. ρ and θ are quantised in the desired resolution. The discrete (ρ, θ) pairs define the bins. Every bin has accumulator. When a given (ρ_i, θ_i) pair gets a vote it's corresponding accumulator is incremented by 1. The **SHT** (Standard Hough Transform) uses one-to-many divergent mapping. This means that every non-zero pixel votes for every possible parameter pair it could belong to. The above mentioned sinusoid is calculated with the desired resolution for the pixel, and the corresponding accumulators are updated.

When the accumulation phase is completed for the whole image, the local maxima of the accumulators are found. Usually a threshold is applied in order to reduce noise and eliminate too short line segments. The radius of the non-maxima suppression also has impact on the results of the line fitting, it must be chosen carefully. After this step the parameters for the most likely line candidates are available.

As the **SHT** does not provide the endpoints of the line, they must be found by examining the original binary image. This can be done by simple checking every pixel along the line with the given parameters and deciding whether or not it is part of the feature. If it is desired, lines with gaps can also be accepted with this method. For more accurate fitting, a Least Squares approximation can also be applied to the pixels belonging to the line.

Progressive Probabilistic Hough Transform

The progressive probabilistic Hough transform is an optimised version of the SHT described in [8]. Probabilistic Hough transform variants were developed to overcome the comparatively high computational cost of the standard transform. The core concept is the same for most probabilistic versions of the Hough transform: not every none-zero point votes, only

a randomly selected subset. These algorithms have to find a balance between minimising the proportion of image points that are used for voting while maintaining the accuracy of the detection process.

The original probabilistic Hough transform[7] solved this issue by introducing a tunable parameter p for the fraction of points to be used. First, a p fraction of the non-zero points are selected, than the SHT is performed on the selected subset. p can be low, the authors of [7] presented successful experiments with $p = 2\%$. However, the results of the algorithm are greatly sensitive to the sampling rate. The authors analysed the problem on the special case of a single line immersed in noise and tried to formulate a solution for determining the p parameter. They succeeded, but the practical applicability is severely limited[8]: it requires *a priori* knowledge of the number of points belonging to the line. There was another approach to calculate the number of necessary votes[2]. It was shown that the probabilistic Hough transform can be formulated as the Monte Carlo approximation of the SHT, thus it is possible to deduce the desired error rate using the theory of Monte Carlo evaluation. Nevertheless, the core problem remained the same: *a priori* information was necessary for determining the sampling rate parameter. Usually there is only very limited information available, so conservative approximation is needed. This leads to the calculation of more votes then necessary, thus reducing the main advantage of the probabilistic method.

The progressive probabilistic Hough transform solves the above issue by "exploiting the difference in the fraction of votes needed to reliably detect lines (features) with different number of supporting points"[8]. This way for long lines only a small fraction of the line's points have to vote for the line to be registered. For shorter lines this proportion is of course higher. For lines with supporting points close to the votes generated by background noise a full transform must be performed.

The authors of [8] proposed the following algorithm to achieve the aforementioned goal. At each iteration a random non-zero image point is selected for voting to the possible model instances it could belong to. After each vote, the question "could the count be due to random noise?"[8] is evaluated. This requires a single comparison per bin update, with a threshold value changing by each vote cast. When a model instance (line) is detected, the supporting points retract their votes. The other points belonging to the same line are removed from the voting process. The pseudo-code representation below is directly quoted from [8].

```
1. Check input image, if it is empty then finish
2. Update the accumulator with a single pixel randomly selected from the
   input image
3. Remove pixel from input image
4. Check if the highest peak in the accumulator that was modified by the
   new pixel is higher than threshold l. If not then goto 1.
5. Look along a corridor specified by the peak in the accumulator, and find
   the longest segment of pixels either continuous or exhibiting a gap not
   exceeding a given threshold.
6. Remove the pixels in the segment from the input image
7. Unvote from the accumulator all the pixels from the line that have
   previously voted.
```

```
8. If the line segment is longer than the minimum length add it into the
   output list.
9. goto 1.
```

This algorithm has some considerable advantages of the standard and other, previous probabilistic variants of the Hough transform. It eliminates the need of *a priori* knowledge necessary for the tuning of probabilistic transforms while it remains much faster than the SHT. It should detect every instance of a model detectable by the SHT, at the latest when the voting finishes with the same number of voted pixels as for the standard transform. Another positive property of the algorithm is that features are detected as soon as the accumulator allows a decision: it is not necessary for all supporting points to vote. The algorithm can also be terminated at any time and still provide some useful output[4].

Originally this transformation method was developed to speed up the Hough transform, while not being considerably more inaccurate. However, an unexpected result was observed by the authors. The PPHT outperformed the SHT in accuracy as well as speed. In sample images consisting of randomly positioned equal length lines, the PPHT produced less false negatives (missed line segments) and less false positives (incorrectly detected lines). This effect is due to the fact that PPHT clears out the votes of the detected lines as soon as they are found. This reduces the clutter in the accumulator, resulting in more accurate results, while also being more computationally efficient.

It also worth noting that the PPHT could, in theory, use every enhancement that were developed for the SHT. For example, the image gradient of the line segments could be used to reduce the number of pixels selected for voting. However, this aspect was not researched in the boundaries of this project.

3.1.2 Line Segment Detector

A fundamentally different approach to line detection was described in [4]. The algorithm is named **LSD** - for Line Segment Detector - by it's creators. It is which, unlike the SHT, detects line segments with subpixel accuracy by default. The runtime of the process is linear in the pixel count of the processed image. It also has fairly good noise suppression. Another attractive property of the algorithm is that it doesn't have any parameters that require tuning by the user. Every one of it's parameters are automatically tuned "under the hood". Because of these advantageous properties was it considered for use in this project. The implementation used was provided by the OpenCV framework. In this section will be a short summary of the theory behind this algorithm.

The LSD takes as input a grayscale image and provides a list of line segments as output. The line detection is based on the image gradient. As a first step, a gradient field is generated

[4]However this aspect is not really important for this project

from the input image. The gradient is taken using a $2x2$ window, see (3.4).

$$\begin{aligned} g_x &= \frac{i(x+1,y) + i(x+1,y+1) - i(x,y) - i(x,y+1)}{2}, \\ g_y &= \frac{i(x,y+1) + i(x+1,y+1) - i(x,y) - i(x+1,y)}{2} \end{aligned} \tag{3.4}$$

Where $i(x,y)$ is the intensity of the grayscale image at (x,y) point. The magnitude of the gradient is calculated by (3.5).

$$G(x,y) = \sqrt{g_x^2(x,y) + g_y^2(x,y)} \tag{3.5}$$

The algorithm uses the angle of the gradient, which will be referred to as LLA (level-line angle), and is calculated by (3.6).

$$\arctan\left(\frac{g_x(x,y)}{-g_y(x,y)}\right) \tag{3.6}$$

The gradient obtained with (3.4) is the image gradient at the point $(x+0.5, y+0.5)$. This half pixel offset is later added to the endpoints of the detected line segments.

Using the gradient information, a level-line field is constructed. Figure 3.4. shows an example of the visualised level-line field. The next step of the algorithm is the segmentation of this field. It happens based on the level-line angle (the gradient angle defined in (3.6)). The pixels that have the same LLA within a given threshold are grouped together. These segments are referred to as *line support regions*, see figure 3.4. for illustration. The segmentation is done with a region growing process.

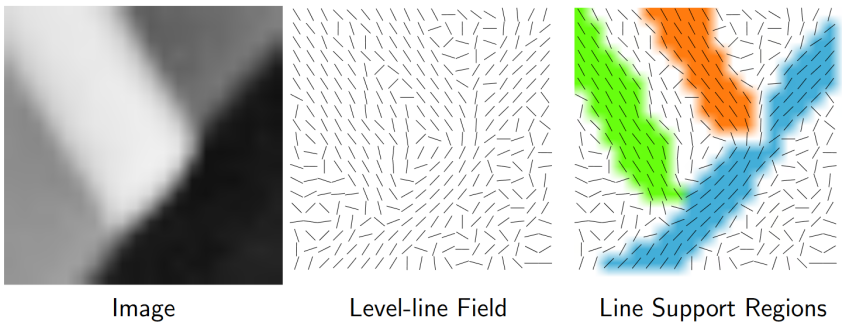

Figure 3.4: *Illustration of the level-line field[4]*

Each *line support region* is a candidate for a *line segment*[4]. The line segments are represented with a rectangle. The main direction of the rectangle is determined by the principal inertial axis of the *line support region*. The size of the rectangle is chosen in a way to cover the whole *line support region*.

The pixels in the rectangle that have LLA close to the angle of the rectangle are called *aligned points*[4]. Figure 3.5. shows an example for the rectangular representation and the *aligned points*. The *aligned points* are used in the validation step of the algorithm.

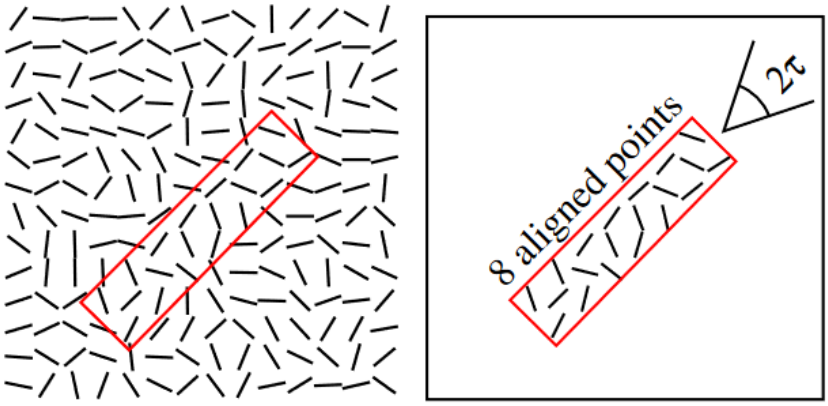

Figure 3.5: *Illustration of the aligned points[4]*

The LSD algorithm uses an *a contrario* validation method. The idea behind that method is checking if it is probable that the current supporting points are caused by random noise. To achieve this, the authors of [4] created a noise model of the **level-line field**. A *line segment* becomes validated if the expected number of it's occurrences on the noise model is low[5].

The algorithm detects the sharp transients in the image gradient. Technically, it detects edges. A line on the image produces two line segments as output, for it's two light-dark transition. The line segments detected by LSD are directional: the order of the endpoints of a line segments depend on the direction of the light-dark transition.

After this short summary of the algorithm[6], some of it's more interesting details will be described.

First off, the algorithm has a preprocessing step. Before calculating the image gradient, the input image is downscaled to 80% along both axes[7]. This is done to cope with aliasing and quantisation artefacts present in most images, for example the staircase effect. The alternative to this subsampling would be the blurring of the image, however that would have some unfavourable side effects. Blurring would affect the statistics of the *a contrario* model. Some structures would be detected in a blurred white noise image. With a correct down-sampling the white noise statistics can be preserved. The choice of scale factor was an optimum between filtering out noise and keeping valuable data.

Another interesting feature of the algorithm is the order in which the possible lines are processed. LSD is a greedy algorithm, it tries to process the most significant edges first. Pixels with higher gradient magnitude correspond to more contrasted edges. In order to process the pixels with the highest contrast first, some ordering is needed. However, most sorting algorithm require $O(n \log(n))$ operations. To avoid this, LSD uses a pseudo ordering that can be done in linear time. The interval between zero and the highest gradient

[5] i.e. It is unlikely to be caused by random noise
[6] The description of the algorithm in pseudo-code form can be found in [4]
[7] or to 64% of it's area

magnitude in the image is divided into 1024 equal bins. Then each pixel is assigned to the bin corresponding to it's gradient magnitude. The processing (region growing) is done first on the pixels selected from the bin containing the largest magnitudes. 1024 levels are enough to almost strictly order the gradients generated from a grayscale image with 256 possible intensities.

To avoid unnecessary processing, a threshold is also applied to the gradient magnitudes. Pixels with a low gradient represent flat regions or slowly changing intensities. These pixels are marked and are not taking part in the later processing steps. This threshold also helps reduce the effects of quantisation noise.

The rectangular approximation of the line segment happens after the segmentation of the level-line field. The rectangle is calculated based on the gradient magnitudes of the pixels belonging to a segment. The gradient magnitude is viewed as the "mass"[4] of the pixel, and the centre of the rectangle is the mass centre point of the segment. The coordinates of the centre point are calculated by the formula given in (3.7)

$$\begin{aligned} c_x &= \frac{\sum_{j \in Region} G(j) * x(j)}{\sum_{j \in Region} G(j)} \\ c_y &= \frac{\sum_{j \in Region} G(j) * y(j)}{\sum_{j \in Region} G(j)} \end{aligned} \tag{3.7}$$

Where $G(j)$ is the gradient magnitude of pixel j, calculated by (3.5). $x(j)$ and $y(j)$ represent the x and y coordinate of point j, respectively. The angle of the main rectangle is defined to be the principal inertial axis of the segment. It can be calculated from the eigenvector of associated with the smallest eigenvalue of the matrix of (3.8).[4]

$$M = \begin{bmatrix} m^{xx} & m^{xy} \\ m^{xy} & m^{yy} \end{bmatrix} \tag{3.8}$$

Where $m^{xx}, \ldots$ is defined below.

$$\begin{aligned} m^{xx} &= \frac{\sum_{j \in Region} G(j) * (x(j) - c_x)^2}{\sum_{j \in Region} G(j)} \\ m^{yy} &= \frac{\sum_{j \in Region} G(j) * (y(j) - c_y)^2}{\sum_{j \in Region} G(j)} \\ m^{xy} &= \frac{\sum_{j \in Region} G(j) * (x(j) - c_x)(y(j) - c_y)}{\sum_{j \in Region} G(j)} \end{aligned} \tag{3.9}$$

This is the short overview of the LSD algorithm, with some of it's more interesting nuances highlighted. The full description is available in [4].

3.1.3 Corner Detection

Detecting quads not necessarily means the detection of line segments. Along with the above described methods based on line detection, a corner detecting algorithm was also benchmarked. The concept of this detection method is as follows. Detect the corners and end-points of a quad with some corner detection algorithm. Checks which detected pairs are connected with lines (or edges). Based on the connected pairs and their ordering, a quad can be reconstructed.

The OpenCV framework provides implementations for some popular corner detection algorithms. Specifically the Harris detector and the Shi-Thomas detector are covered. In this section will be a short theoretical summary of corner detection in general, and some specifics of the above mentioned solutions.

The basic idea of most corner detection algorithm is the following. Considering a local window on an image, corner regions show large change in average intensity if the window is shifted by a small amount in any direction. The mathematical formulation of the following idea is shown in (3.10). $E_{x,y}$ is the change in intensity produced by shifting the window by (x, y). $I_{u,v}$ is the intensity of the image at the point (u, v), and $w_{u,v}$ specifies the image window. In the simplest case, the image window is rectangular and it is unity in a specified region and zero otherwise.

$$E_{x,y} = \sum_{u,v} w_{u,v} |I_{x+u,y+v} - I_{u,v}|^2 \tag{3.10}$$

A naive approach of corner detection is to use (3.10) as it is. The local maxima of the minimum of (3.10) above a certain threshold can be used for a metric. With this method, three cases have to be considered:

1. **Flat region:** The windowed image region has almost constant intensity. In that case, all shifts will show small change.
2. **Edge region:** The windowed image region has an edge in it. In that case shift in one direction will result in large change, but shifts in other directions will show low change in intensity.
3. **Corner region:** If the windowed region contains a corner, all shifts will show large change in the intensity.

The shifts can be chosen in a couple of ways: 90° shifts, 45° shifts in 8 or 4 directions, etc... Actually, this detection method is analysed in [5], and it is the base of the Harris detector.

However, the the above described corner detector suffers from a number of problems[5]. Firstly, it provides an anisotropic response, as only a discrete set of shifts are used. To

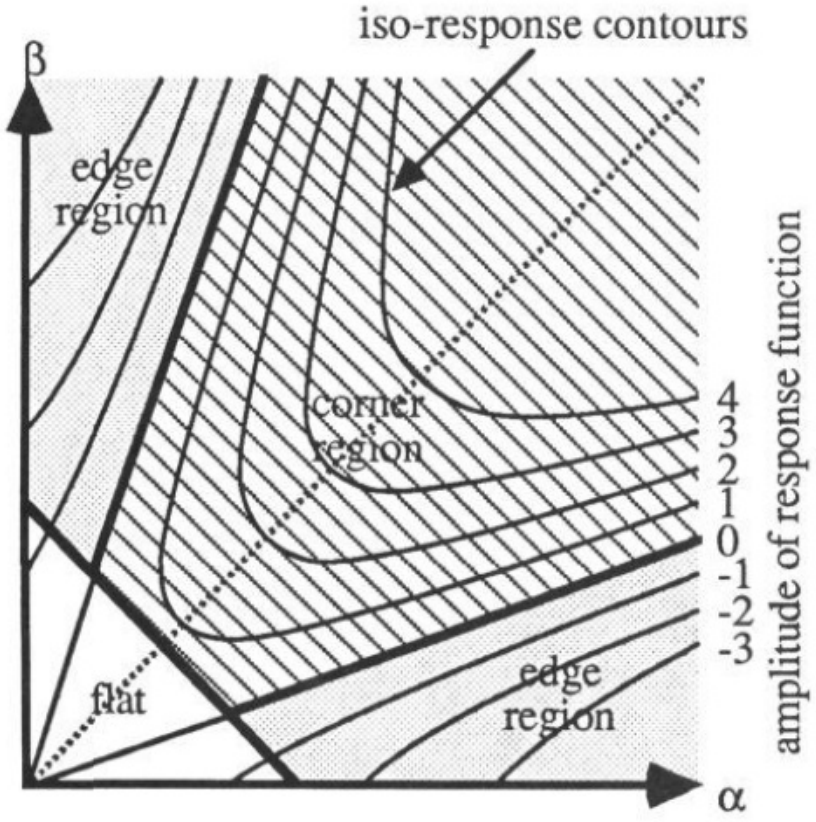

Figure 3.6: *Image point classification based on Harris measure[5]*

address this issue, the Harris detector uses an analytic expansion of (3.10) around origin. See (3.11).

$$E_{x,y} = \sum_{u,v} w_{u,v}(I_{x+u,y+v} - I_{u,v})^2 = \sum_{u,v} w_{u,v}(x\frac{\partial I}{\partial x} + y\frac{\partial I}{\partial y} + O(x^2, y^2))^2 \tag{3.11}$$

Another problem is that the above detection method's response is noisy[5] because of the rectangular and binary image window. The Harris detector resolves this issue by using a circular and smooth (for example Gaussian) window:

$$w_{u,v} = e^{-\frac{u^2+v^2}{2\sigma^2}} \tag{3.12}$$

The third issue Harris found with the use of (3.10) as a corner metric is that it responds too readily t edges[5]. This is because only the minimum of E is taken into account when deciding whether a sampled window contains a corner or not. To address this issue, a reformulation of the corner measure was proposed that takes the variation of E with the direction of the change into consideration. For small changes, E, the average change in intensity generated by a shift with (x, y) can be written as:

$$E(x, y) = (x, y)M(x, y)^T \tag{3.13}$$

Where M is composed of the image gradients in the window. (3.15) shows M using the

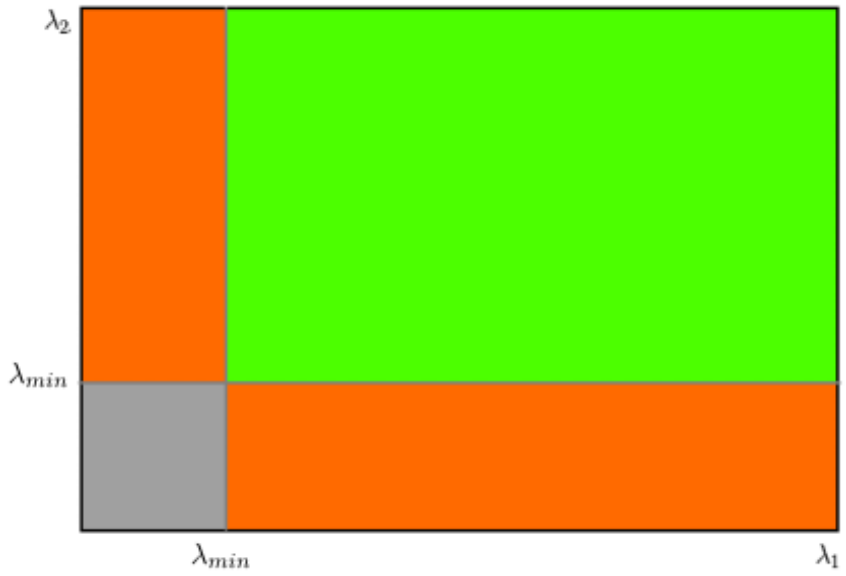

Figure 3.7: *Image point classification based on Shi-Tomasi measure. Image source: OpenCV documentation*

notation of (3.14)

$$I_x = \frac{\partial I}{\partial x}, I_y = \frac{\partial I}{\partial y} \tag{3.14}$$

$$\begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \tag{3.15}$$

Note that M describes the shape of the local autocorrelation function's shape at the origin. To describe E, [5] uses the eigenvalues of the matrix M, as it provides a rotationally invariant description. With this new method, the above mentioned cases (flat region, edge, corner) can be expressed as follows.

1. **Flat region:** Both eigenvalues are small.
2. **Edge region:** One eigenvalue of M is small, the other is comparatively large.
3. **Corner region:** Both eigenvalues are large.

Figure 3.6. shows the above defined regions with respect to the eigenvalues (α and β are the eigenvalues of M). The Harris detector also uses a metric for the "quality" of the detected edges or corners. This is noted with R and is defined below.

$$R = \det(M) - k(Trace(M))^2 \tag{3.16}$$

k is a tunable parameter, its value is usually in the range of $0.01 - 0.05$. The higher R is, the more likely that a corner is present in the sampled window. In figure 3.6. the curves mark the regions in the $\alpha - \beta$ space that have the same R value.

This is the short theoretical summary of the Harris detector. A detailed mathematical derivation of the formulae can be found in [5].

The other corner detection method provided by the OpenCV framework is the Shi-Tomasi detector[9]. This detector uses the same concept as the Harris detector, however uses a different measure to classify features as corners. The scoring function is defined in (3.17). λ_1, λ_2 is the same as α, β for the Harris detector.

$$R = \min(\lambda_1, \lambda_2) \tag{3.17}$$

A feature is classified as a corner if R is greater than a threshold λ_{min}. Figure 3.7. shows this relation in the $\lambda_1 - \lambda_2$ space. The Shi-Tomasi measure, while being simpler and requiring less computational power, shows better performance in images[9].

3.2 Application for Quad Detection

In this section the practical application of the previously explained algorithms will be presented. The detection methods described in the above section provide solutions for detecting geometric primitives. Some additional processing is required to use them for identifying quads on an image and measuring their parameters. The details of the additional logic differ from algorithm to algorithm. The preprocessing steps, the conversion of geometric primitives to quads and the various other necessary tasks will be explained below. For the sake of simplicity, the steps of quad detection will be shown using a pseudo-code notation.

The experimental part of this work was done in python. The quad detector prototypes and the test framework was developed using the language. This greatly influenced the design of the framework used for testing: object oriented design principles were followed. The quad detection routines were encapsulated in *quad detectors*, which in turn are represented as instances of a *QuadDetector* class. Each detection method had it's own class, derived from the common ancestor. The implementation details of the classes will not be described in depth, as they are only part of the test framework. Only the parts necessary to understand the test method will be shown.

Each implemented detector had, as a public interface, a function called *detect_ quad(img)*, which required an image containing a single quad. The test framework operated on this simple interface. The pseudo-code blocks of the following subsections will show how each detection method implemented this interface function.

As mentioned above, the exact python implementations will not be discussed. They are far from optimal and there is lots of room for improvement. However, in order to be complete, they are published in the appendix.

3.2.1 LSD Quad Detector

This quad detection method is built around the LSD algorithm. The OpenCV implementation is used, which provides additional improvements around the algorithm described in

the theoretical overview. Namely, the scale dependence is handled within OpenCV. It is done by generating a Gaussian pyramid from the input image, with $N-1$ down-sampling and N Gaussian blurring, resulting in N layers (one for each octave). The above described LSD algorithm is then ran on each layer to detect lines of multiple scales.

That being said, not much additional logic was necessary to build a quad detection solution. The detection routine is summed up in the listing below.

```
detect_quad(img):
  img = prepare_image(img)
  lines, widths, prec, nfa = LSD_detector.detect(img)

  # Both edges of some line segment found
  if count(lines) > 3:
    pairs = find_pairs(lines, widths)
    lines_merged = merge_pairs(pairs, widths)
  elif count(lines) == 3:
    lines_merged = lines
  # Detection failed
  else:
    return None

  corners, result = find_corners(lines_merged)
  if result == (BaseNotFound or IntersectionNotFound):
    # detection fail
    return None

  return Quad(scale_to_quad_space(corners))
```

This line detection algorithm requires very little preprocessing. The only necessary step is converting the 3 channel input image to grayscale. Noise filtering is not needed as it is "built in" to the underlying algorithms. First, the Gaussian down-sampling provides a quite efficient protection against white noise, jagged image edges or JPEG artefacts. Furthermore, the *a-contrario* validation of the LSD algorithm also provides efficient noise suppression.

It is possible that the LSD detects both edges of a line. In that case, they need to be merged. This is done by looking for almost parallel lines which are close together. The merged line segments are calculated by selecting the longer one of the two parallel segments. Then it is offset by half of the width of the detected line. The LSD algorithm is sensitive to the gradient direction, so the direction of the offset can be calculated.

After the duplicates have been dealt with, the corner coordinates of the quad need to be found. In order to do this, the quad base has to be identified. The identification begins with creating every possible combination of the three line segments. Then the distance of the line segments is calculated for each pair. The pair with the maximum distance contains the the quad *arms*, and the remaining segment is the *base*. The inner quad corner coordinates are found by calculating the intersection of each **line** representing an [arm] with the one **line** representing the *base*. The outer corners (the end of the arms not connecting to the base) are simply the endpoints of each arm **line segment** further from the base. This corner detection method is shared by all quad detectors using some kind of line detection.

Finally the corner coordinates are normed by the image size. This is done in order to store the quad parameters in a way that does not depend on the image size.

3.2.2 Hough-based Quad Detectors

There are two quad detectors based on variants of the well known Hough-transform. One is based on the Standard Hough-transform, the other is based on the PPHT. Both versions of the transforms were covered in the theoretical overview. The two detectors share many implementation details, so the common parts will be discussed first. The different aspects will be covered separately.

The general flow of the Hough-based detection algorithms is given by the following pseudo-code. The different underlying line detection methods are hidden behind the *find_ segments* function. The separate implementations will be discussed later.

```
detect_quad(img):
  img = prepare_image(img)

  line_segments, result = find_segments(img)
  if result == NoLinesDetected:
    # Detection fail
    return None

  if count(line_segments) < 3:
    # Detection fail
    return None

  corners, result = find_corners(line_segments)
  if result == (BaseNotFound or IntersectionNotFound):
    # Detection fail
    return None

  return Quad(scale_to_quad_space(corners))
```

The detection process starts by preparing the input image for the Hough-transform. This involves two steps. First the image is converted to binary by applying a threshold to it. From the binary image the skeleton of the feature is extracted. For this a simple algorithm is used, based on morphological operations.

The line detection is carried out on the skeleton image. The detectors return the detected line segments. Then the corner extraction described with the LSD based detector is applied.

After this overview of the general detection algorithm, the specialised cases of the standard and probabilistic Hough transform will be described.

SHT Quad Detector

The Hough transform is used in the line segment detection step. It's implementation is provided by the OpenCV framework, but there are some supporting logic to get useful

output from the detected lines. The OpenCV function *HoughLines* returns lines, represented by the *Rho-Theta* parameters described in the theoretical overview. It is not an easy task produce only the wanted lines with the classical Hough transforms. Based on the above points, the supporting logic has to do two things. First, the relevant lines need to be selected, then the endpoints of the lines have to be determined.

The line extraction routine is described in pseudo-code below. The resolution of the accumulator was set to 1 pixel for *rho* and 1° for *theta*. The threshold for detection is determined by the bounding box of the skeleton on the image, as it correlates with the length of lines expected to be found.

```
find_lines(img):
  thresh = get_threshold(img)
  lines = HoughLines(skeleton, 1, pi / 180, thresh)
  if lines is None or count(lines) < 3:
    return NoLinesDetected

  if count(lines) > 3:
    lines = kmeans(lines, number_of_clusters=3)

  return lines
```

The problem of more than 3 lines found is handled as follows. The lines returned by the transform are clustered by the k-means algorithm. The cluster centers returned by k-means each correspond to the average of one of the 3 dominant line groups.

As for the creation of line segments from the lines, the following algorithm is used. First, the pixel from the original image touched by the given line are collected. Then, the longest continuous[8] range of non-zero pixels are marked as part of the line segment.

```
find_segments(img):
  lines, result = find_lines(img)
  if result == NoLinesDetected:
    # Detection failed
    return result

  max_line_gap = min(bounding_box_size) / 2
  segments = empty_list
  for rho, theta in lines:
    line_points = get_line_points(rho, theta, img)
    segments.append(get_line_segment(line_points, max_line_gap)

  return segments
```

The two ends of the region os the so marked pixels represent the endpoints of the line segment. Then a list of the found line segments is returned.

PHT Quad Detector

The probabilistic Hough transform implementation is also provided by the OpenCV framework. It requires additional parameters, as it handles the line to line segment conversion internally.

[8] that is, having a smaller gap than defined by the max_line_gap variable

The PHT-based implementation of *find_segment* is described by the following pseudo-code. The threshold for detection is calculated from the feature bounding box similarly to the SHT-based detector, the only differenc is a scale factor.

```
find_segments(img):
  thresh = get_threshold(img)
  min_line_length = min(bounding_box_size) / 4

  lines = HoughLinesP(img, 1, np.pi / 180, thresh,
                      minLineLength=min_line_length,
                      maxLineGap=min_line_length/2)
  if lines is None or count(lines) < 3:
    return NoLinesDetected

  if count(line_segments) > 3:
    line_segments = merge_segments(line_segments)

  return line_segments
```

For the minimal required line length a simple heuristic is given, also based on the bounding box.

This detection method also suffers from the problem of more than necessary lines detected. It is solved by merging the line segments having similar slope. The detected line segments are returned to the common part of the code for quad corner calculation.

3.2.3 Corner Quad Detector

The corner detection based algorithm is fundamentally different than it's line detection based alternatives. It is also simpler, as there is no line segment to corners conversion. This detection method also has 2 variants depending on which corner measure they use. However, this is handled inside OpenCV, so the variants differ only in a passed parameter.

The detection method overview is shown below. The image preprocessing involves only a conversion to grayscale. This could be improved, but for the preliminary testing it wasn't necessary, as optimally generated images were used.

```
detect_quad(img):
  img = prepare_image(img)
  corners = get_corners(gray, max_corner_count=6,
                               quality_level=0.1,
                               min_distance=2,
                               methotd=shi_tomasi)

  if count(corners) < 4:
    # Detection failed
    return None

  if count(corners) > 4:
    corners = merge_corners(corners)

  inner, outer, result = identify_points(corners)
  if result == (BaseNotFound or TypeError):
    # Detection failed
```

```
    return None

corners = [outer[0], inner[0], inner[1], outer[1]]
return Quad(scale_to_quad_space(corners))
```

The corner detection function was provided by OpenCV. As mentioned above, the scoring function can be set to either Harris or Shi-Tomasi. Both were tried, but there was no noticeable difference in the quality of detection. Thus, the Shi-Tomasi method is used, as it is more efficient computationally. The maximum number of corners to detect was seto to 6: the 2 endpoints of the arms, and the inner and outer corners of the intersection of base and arms.

The usual error handling is also present here: if less than 4 corners are detected, detection failure is reported. If more (up to 6) corners ar detected, corner merging is done. This is necessary if both the inner and the outer corner of the base-arm intersection is detected. Those are merged together to their average. For now, always the 2 closest points are merged.

In order to create a quad, the corner points have to be identified as an inner (base-arm intersection) or outer (arm end) points. To do this, the connectivity of the point pairs is examined. The connectivity check is done by counting the non-zero pixels along the line segment between the two selected corner point. The points with 2 neighbours are classified as inner points, the ones having only one neighbour are the outer points.

When this classification is done, the detection process is finished. A quad is created from the scaled corner points.

3.3 Performance comparison

The major part of this work is dedicated to selecting the most suitable algorithm for measuring quad parameters. So far the theory of the underlying algorithms and the minimum viable implementations of detectors based on them have been discussed. In order to select the best candidate some kind of benchmarking solution is needed. Defining a scoring function to be used as a base for comparison was a task to be solved. Also, to draw any meaningful conclusion a significantly large number of test need to be run.

To address these tasks, a test framework was developed. Similarly to the detectors themselves, it was also done in python. The software handles the whole benchmarking process. It generates quads for test data, renders them to images, runs the detection routines, plots the test result, etc... This test framework is described with the necessary level of detail in this section.

It was also a non-trivial decision to select a scoring function for the comparison. Below will be a summary of the tried and used error functions. The reasons for their consideration as well as their definitions will be explained.

The last, major part of this section will be the discussion of the test results. Each algorithm will have a thorough analysis about it's strengths and weaknesses, and of their error distribution. Finally, the summary of the results will be published, comparing the results of the detectors. A recommended quad detection method will be selected.

3.3.1 Test method

In this project multiple quad detection solutions are prototyped. The best is needed to be selected. It is not a trivial choice, as there are many criteria to be satisfied. The algorithm needs to be accurate, robust, relatively fast, work on noisy images, etc... Thorough testing is necessary to select the best algorithm for the task. The tests should be reproducible and statistically relevant. In this section will be a description of the testing method used in this project.

As a first step, a data source is necessary for the algorithms under test. For this, randomly generated quads were used. The random generator was constructed in a way that allowed some control over the generated data: the *quad size* and the range of the other parameters could be set. To provide data for extensive testing, a large data set was generated. Every algorithm received random quads with sizes ranging from 1% to 75% of the image size, in 1% increments. From each size 1000 quads were generated and provided to the detectors. These values were chosen to cover a significant portion of the parameter space and to provide statistically significant results.

The generated quads were rendered by OpenCV and the generated images were the inputs for the detection algorithms. The testing was done with images containing only a single

quad. There are many reasons for this. First, it is important to limit the test scope. In this step the quad detection was benchmarked. not the segmentation logic. In the real use-case, the input image containing a marker is segmented, and the segments are fed to the quad detector separately. The images were rendered with different levels of additive Gaussian White Noise. The test series were run first on optimal images, and after that rerun with more and more added noise. This was done to test the robustness. Although the GWN covers only a small portion of the noise in a real image, these test did provide some insight.

The accuracy of the detection was analysed based on many different error measure. Those are defined in the next section. The experiments measured the expected value and the standard deviation of error of the algorithms. These were plotted at the end of the test cycle.

The test method is summarised by the following pseudo-code program.

```
quad_groups = empty_list
for size in [0.01:0.01:0.75]:
        group = generate_quads(count=1000, size=size)
        quad_groups.append(group)

error_series = empty_list
for group in quad_groups:
        pairs = empty_list
        for quad in group:
                img = render(quad)
                img = add_noise(img, noise_level)
                detected_quad = detector.detect_quad(img)
                pairs.append(quad, detected_quad)
        group_error = calculate_error(pairs)
        error_series.append(group_error)
display_result(error_series)
```

The quads were rendered to $640 * 640$ *pixel* images. The quad sizes (base lengths) are normed to image size, that is $640px$. A quad with size 1 would have a 640 pixel long base. The same scaling applies to the diagrams of the following sections.

3.3.2 Error measure

In order to compare the performance of the quad detection algorithms some kind of scoring function is needed. For this the use of detection error seems intuitive. As a quad has 6 independent parameters, and these are stored in 2 different representations, defining an error measure is not straightforward. Because of this, several scoring functions were defined during development, and many of those are actually used for comparing the algorithms.

As described in the previous section, the calculation of detection error is based on rendering a known quad and running the detection algorithms on it. After the detection is done and it is successful[9], some kind os measure is necessary to calculate the "distance" of the detected and the original quad. Below will be the definitions the error measures used within this

[9] A quad instance is returned, which is not guaranteed

project to compare the quad detection algorithms. The notations used in the formulae are listed here.

- Q^o, Q^d: Original quad, detected quad
- Q_p: Parameter p of quad Q, where p can be any of the followings: $\{s, m_a, m_b, \alpha, \beta, \gamma\}$.
- C^o, C^d: Corner set of the original and the detected quad
- C_i^o, C_i^d: The $i.$ corner of the original and the detected quad
- $C_{i,x}$: x coordinate of the $i.$ corner of a quad
- P^o, P^d: the parameter space of the original and the detected quad. Using Section 2.1's notation, it can be defined as $P = \{s, m_a, m_b, \alpha, \beta, \gamma\}$

As mentioned above, there are two different quad representations used[10]. One stores the coordinates of the corners, the other the quad parameters. The most intuitive way for error calculation is based on the corner representation. We can calculate the distance between the detected and the original corner. (3.18), (3.19) and (3.20) define 3 scoring functions based on this idea.

$$E_{c,abs} = \sum_{1}^{4} \sqrt{(C_{i,x}^o - C_{i,x}^d)^2 + (C_{i,y}^o - C_{i,y}^d)^2} \tag{3.18}$$

The first possibility is to calculate the sum of the distance between the detected and the original corners. This naive approach has many drawbacks. The main concern is that it does not take into account the *size* of the quad. The cumulation of error from every corner also distorts the results. In reality, if every corner has some amount of noise in it's position, the quad parameter representation can still be quite close to the original. However, if only one corner has a larger amount of error, the distortion will be much higher. This error measure reports the same amount for both cases.

$$E_{c,avg} = \frac{1}{4} E_{c,abs} \tag{3.19}$$

The above points are also true is the average of the absolute displacements are used.

Better results can be achieved by using relative coordinate error. The formula used by this work can be seen in (3.20). The problem of the error depending on the quad *size* is solved by this. As seen on the formula, the coordinate error is compared to the coordinates of the original quad corners.

$$E_{c,rel} = \frac{1}{4} \sum_{1}^{4} \frac{\sqrt{(C_{i,x}^o - C_{i,x}^d)^2 + (C_{i,y}^o - C_{i,y}^d)^2}}{\sqrt{(C_{i,x}^o)^2 + (C_{i,y}^o)^2}} \tag{3.20}$$

[10] for details, see Section 2.1

(3.20) was chosen for the comparison of algorithms based on the accuracy of corners detected.

The following scoring functions are based on the other quad representation. This has considerable advantages compared to the corner based representation, because a much clearer picture of the distribution of error factors is obtained. The quad parameters can be classified into 3 sets. The is the quad size, which is an absolute length, measured in pixels. Another category is formed of the angle parameters: the angle between the *base* and each *arm*, and the orientation (which is basically the angle between the *base* and the image frame). The third is the multiplier parameter. These are not absolute lengths, they are calculated based on the base length. Below will be proposed error measures for the three categories.

For the angle parameters, as a first approach an absolute error measure was defined. As (3.21) shows, the 2 angle errors are summed. As an alternative, their average also can be used. Contrary to the coordinate errors, here the absolute angle error does not depend on the size of the quad. These scoring functions can be useful if the absolute magnitude of the error is of interest.

$$E_{a,abs} = |Q^o_\alpha - Q^d_\alpha| + |Q^o_\beta - Q^d_\beta| \tag{3.21}$$

$$E_{a,avg} = \frac{1}{2} E_{a,abs} \tag{3.22}$$

Of course, relative error can also be defined for the angles, too. See (3.23). This was used for comparison, because the relative values given as percentages were easier to evaluate.

$$E_{a,rel} = \frac{1}{2}\left(\frac{|Q^o_\alpha - Q^d_\alpha|}{|Q^o_\alpha|} + \frac{|Q^o_\beta - Q^d_\beta|}{|Q^o_\beta|}\right) \tag{3.23}$$

The multipliers are very similar to the angles in terms of error metrics. They describe a relative parameter, so the quad scale is not an issue here. Nonetheless, both absolute and relative error measures were defined for completeness. Similarly to the angles, both the sum os absolute errors and their average can be meaningful, depending on what is the goal of analysis.

$$E_{m,abs} = |Q^o_{ma} - Q^d_{ma}| + |Q^o_{mb} - Q^d_{mb}| \tag{3.24}$$

$$E_{m,avg} = \frac{1}{2} E_{m,abs} \tag{3.25}$$

For readability, the relative measure was chosen as a basis for comparison. The values are normed with the respective parameter of the original (generated, thus it's parameters are known to an arbitrary level of precision) quad.

$$E_{m,rel} = \frac{1}{2}\left(\frac{|Q^o_{ma} - Q^d_{ma}|}{|Q^o_{ma}|} + \frac{|Q^o_{mb} - Q^d_{mb}|}{|Q^o_{mb}|}\right) \tag{3.26}$$

Orientation is handled differently from the other angle parameters. This is due to two reasons. First, it's conceptually different. It describes a rotation as opposed to α and β

that describe angles between lines segments. The second reason is mainly empirical: it turned out to be less error-prune than the other two.

$$E_{o,abs} = |Q^o_\gamma - Q^d_\gamma| \tag{3.27}$$

$$E_{o,rel} = \frac{|Q^o_\gamma - Q^d_\gamma|}{|Q^o_\gamma|} \tag{3.28}$$

Similarly to the previously discussed categories, the absolute and relative errors are defined. Equations (3.27) and (3.28) show the definitions. For comparison, as before, the relative error was used.

The *base length* or *size* is the only absolute length parameter in this quad representation. It makes sense to handle it separately, because many other parameters (orientation, arm multipliers) depend on it. The scoring functions are defined below, similarly to the previous ones.

$$E_{o,abs} = |Q^o_s - Q^d_s| \tag{3.29}$$

$$E_{o,rel} = \frac{|Q^o_s - Q^d_s|}{|Q^o_s|} \tag{3.30}$$

For consistencies sake, also the relative error was used here for comparison.

The above defined error functions provide useful information if the distribution of error between the quad parameters is interesting. However, this parametric quad representation lacks a scoring function that would provide information on the error magnitude as a whole. To resolve this, one more error measure is proposed and used by this work. The independent parameters of a quad can be viewed as coordinates in a 6 dimensional space. The analogy stands, as the parameter space is a subset of $\mathbb{R}^6$. A distance in that 6-dimensional space can be defined, see (3.31).

$$E_{sum} = \sqrt{\sum_{p \in P^o, q \in P^d} (p - q)^2} \tag{3.31}$$

This gives useful information about the absolute magnitude of error in the parametric quad representation. Of course, the relative version of the above error measure can also be defined.

$$E_{sum,rel} = \frac{\sqrt{\sum_{p \in P^o, q \in P^d} (p - q)^2}}{\sqrt{\sum_{p \in P^o} p^2}} \tag{3.32}$$

With this, a complete set of error measurement functions are defined.

It is reasonable to use both quad representations, thus both sets of error measures. The corner representation is intuitive. Also, as it is stated in the previous chapters of this work, the pose estimation algorithms use corresponding point pairs. So the most relevant error component seems to be the one present in the quad corner coordinates, as it directly influences the accuracy of the calculated camera pose.

However, the two representations are equivalent. It is possible to convert between the two without loosing useful information. If discrete markers are used, with the parametric

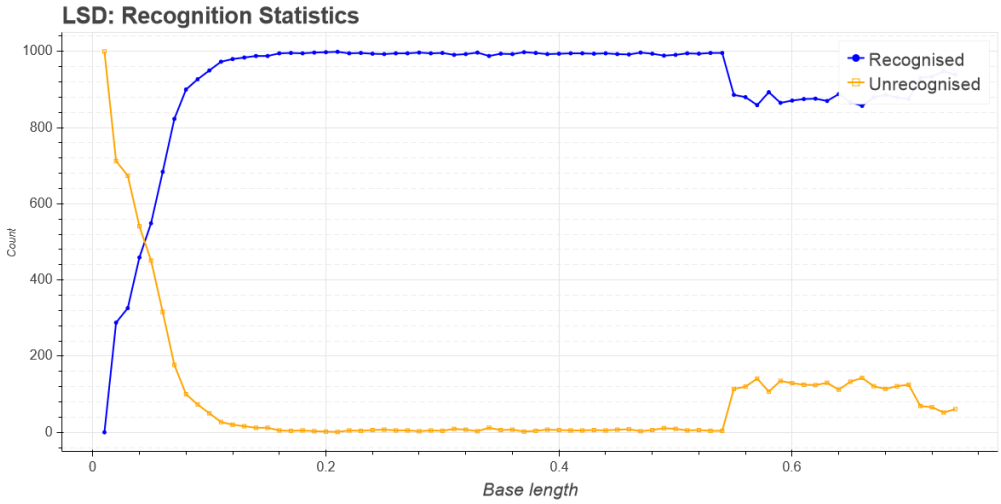

Figure 3.8: *Recognised and unrecognised quads with respect to quad size, using the LSD method*

representation it is possible to further refine the detection results. This can be done by replacing the detected quad with the closest known discrete one[11]. Currently this aspect of the marker is not implemented, it can be a subject of future improvements.

3.3.3 LSD Quad Detector results

In this section the performance of the Line Segment Detector-based quad detection method will be evaluated. Figure 3.8. shows the recognition count of the LSD quad detector. Recognition fail is reported by the test framework if the detector couldn't find a quad on the input image or the average relative error is greater than 100%. Detection failure can be caused by a couple of reasons within the algorithm. If the quad on the image is too small, less than 3 lines may be detected. The detector is helpless in this case, no quad is returned. Another failure cause is connected to the small quads. If the necessary number of line segments are detected, it is still possible for the detection to fail. It can happen because a small quad rendered in comparatively low resolution looks more like a blob than connected line segments, thus the detected segments are not necessarily correspond to the segments of a quad. These effects combined explain the rising edge in the beginning of the detected quad count on figure 3.8..

The sudden increase in the unrecognised count in the region of larger quads can be caused by a combination of two effects. The quad rendering process is designed in a way that larger quads are rendered with thicker lines than the small ones. This can cause issues with the merging of the two edges into one line. The detector algorithm can be improved to better handle this. The other part is that the LSD algorithm is sensitive to the scale of the features compared to the image size. Detection can be improved if the algorithm is re-run on a down-sampled image.

[11] This will be elaborated later on. This sentence is just to give a general idea and is not technically correct.

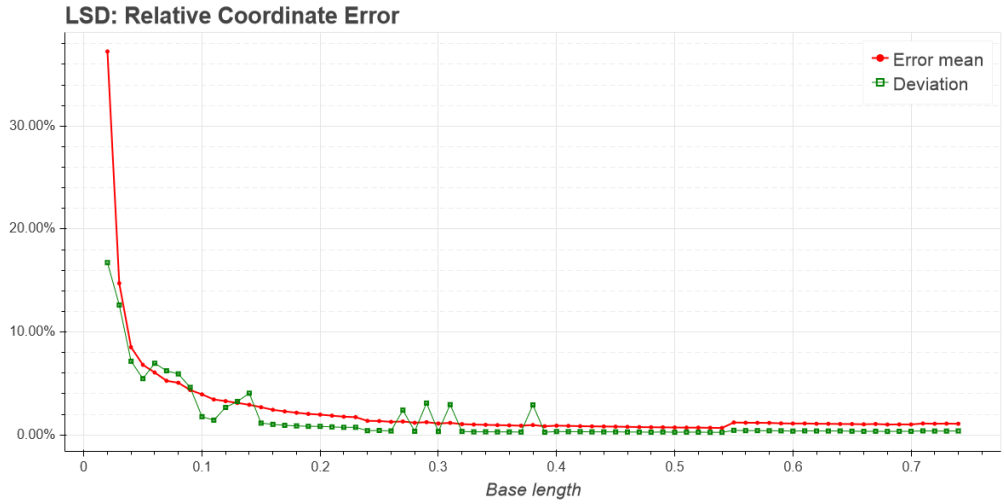

Figure 3.9: *Relative coordinate error with respect to quad size, using the LSD method*

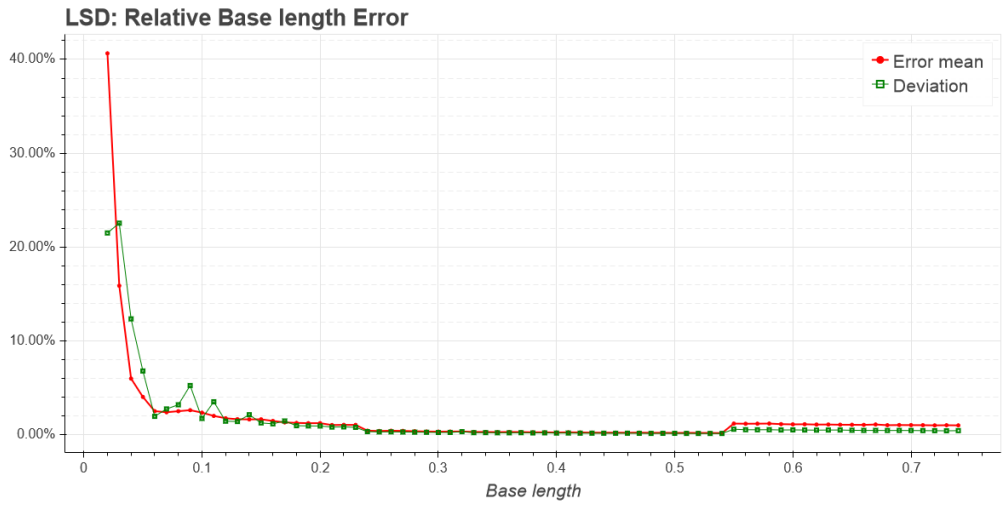

Figure 3.10: *Relative base length error with respect to quad size, using the LSD method*

The overall performance of the LSD quad detector is quite good. Figure 3.9. shows the average relative coordinate error and it's deviation as a function of the quad size. This figure can be used as an indicator of the general performance the algorithm. The relative coordinate error is below 5% before the 0.1 scale factor is reached. This means that the algorithm can detect quite small features relatively accurately. In the optimal case, the average error is below 1%. As the figure shows, the deviation of the error is also small, which indicates that the detection is stable.

The rest of the quad parameters show roughly the same level of accuracy. The angle error (figure 3.12.) and the multiplier error (figure 3.11.) stay below 5% for most of the usable size range.

The error in the detected coordinates, base length, and multipliers show a small increase toward the region of larger quads. This is caused by the same effect as causes the increase

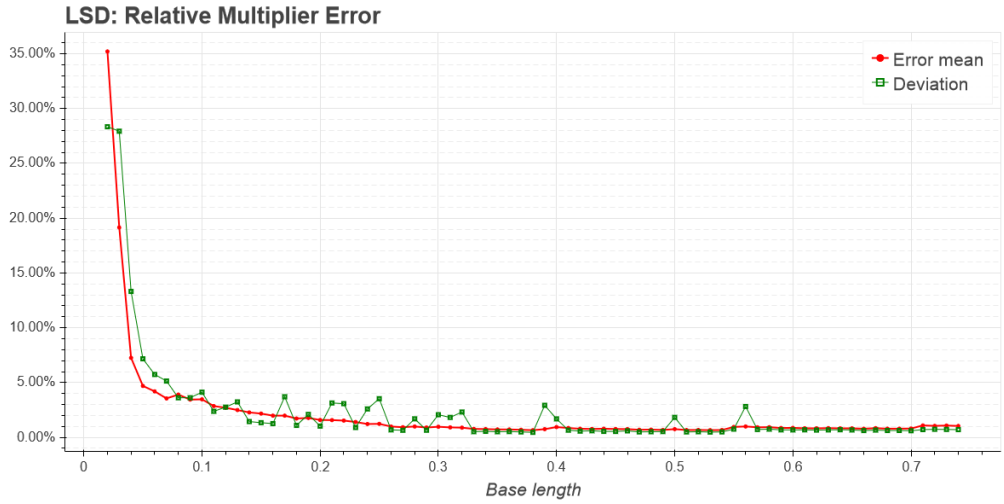

Figure 3.11: *Relative multiplier error with respect to quad size, using the LSD method*

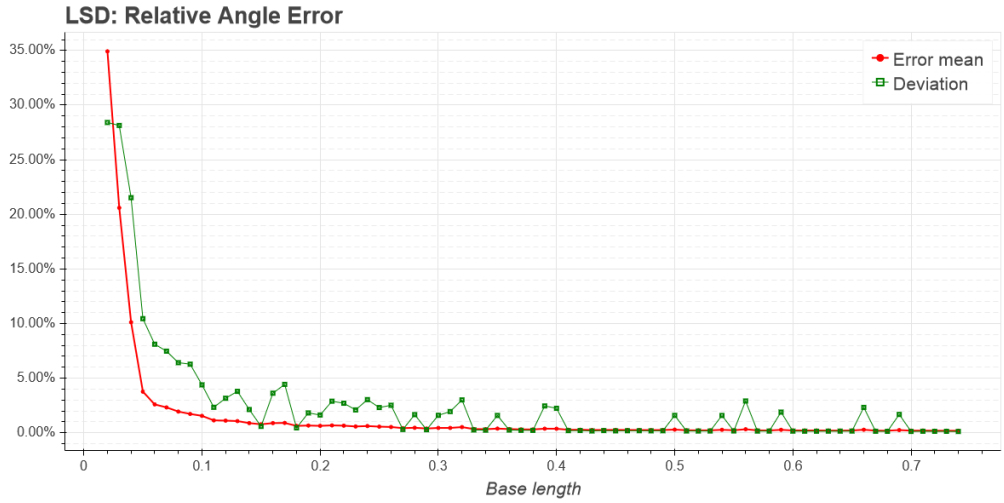

Figure 3.12: *Relative angle error with respect to quad size, using the LSD method*

in recognition failure. However, the angle parameters are not affected by this issue. This is not unexpected, as the angle between lines is left unchanged by the offset caused by the line width.

Another interesting observation can be made about the error distribution. Figure 3.10. and 3.13. show a significantly smaller error level than the others. Both the base length and the orientation only depends on the detection of the quad base, which is usually the largest feature of a quad. This results in a more accurate detection, as more inliers are available for precise line detection. The orientation is the most accurately detectable quad parameter. On top of it depending only on the quad base, it is unaffected by the error in the detection of the base length. Only the angle of the line is important.

Although the runtime of the algorithms was not measured during the experiments, the LSD method was noticeably faster than the others. For any meaningful statistics on runtime,

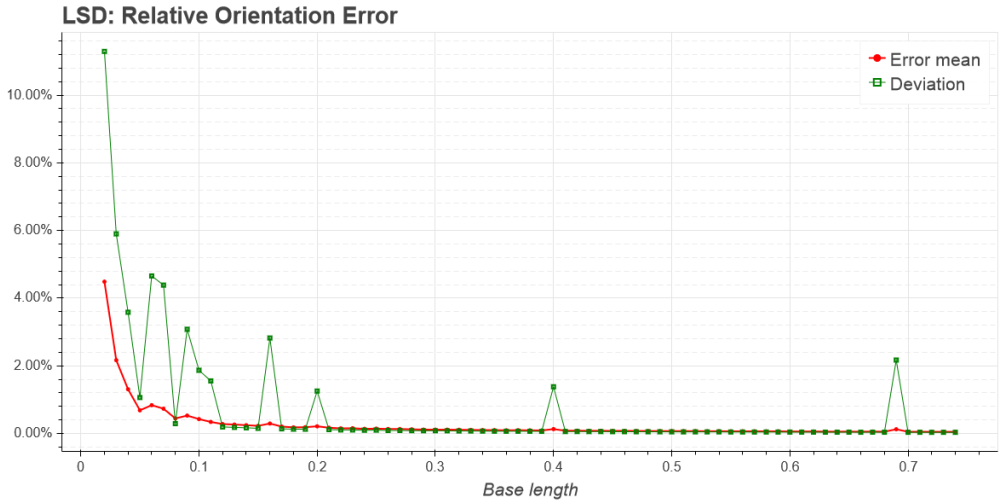

Figure 3.13: *Relative orientation error with respect to quad size, using the LSD method*

an equally well optimised, preferably C++ implementation of the algorithms would be necessary. This was not in the scope of this project.

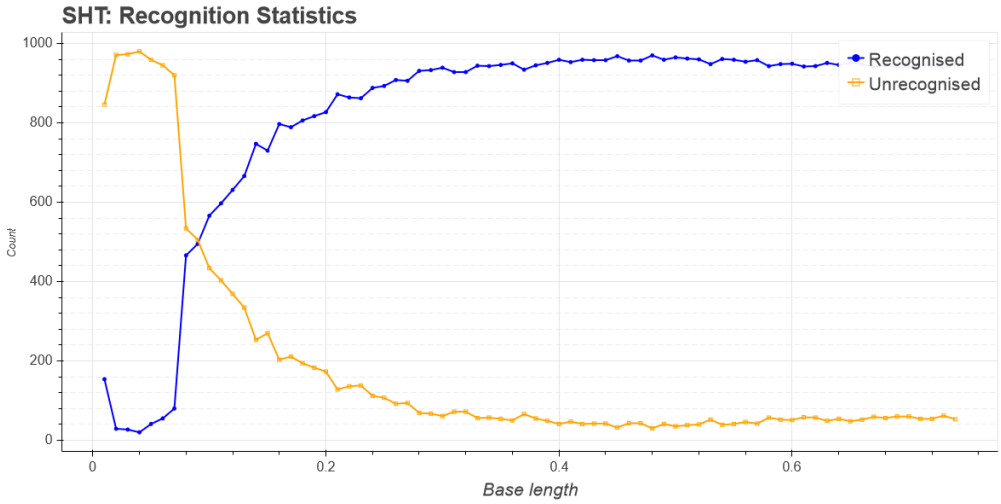

Figure 3.14: *Recognised and unrecognised quads with respect to quad size, using the SHT method*

3.3.4 SHT Quad Detector results

This quad detection method is built around the Standard Hough Transform, with all of it's well known benefits and drawbacks. It provides usable results, but there are better alternatives amongst the tested detectors. Below is the performance analysis of the SHT-based detector.

Figure 3.15: *Relative coordinate error with respect to quad size, using the SHT method*

Figure 3.14. shows the recognition statistics of the detector. It needs much larger quads to work, compared to the other detectors. This is partly caused by the SHT's need to have a comparatively large number of inliers supporting a line to detect it. For small quads, some line segments can be simply shorter than what is detectable by the algorithm. Or false line can be detected, which also makes the quad reconstruction impossible.

The Hough transform has tunable parameters, which greatly affect what features can be

found. However, there is no formula to determine the values of these optimally based on the image parameters, so some heuristics are necessary to guess them. The detection rate could possibly be optimised by tweaking that heuristics, but it is outside the scope of this project. For now, it is accepted as a limitation that the SHT-based quad detector provides useful information above the 0.2 quad size.

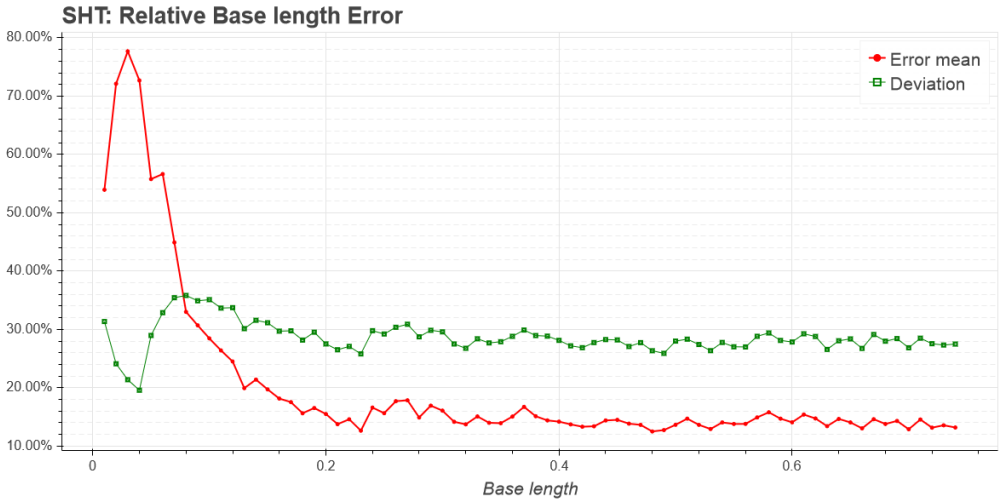

Figure 3.16: *Relative base length error with respect to quad size, using the SHT method*

Figure 3.15. shows the average relative coordinate error of the algorithm. It is much larger than that of the LSD quad detector. The error never goes below 15%, not even for large quads. As stated above, the detector in it's current state is not recommended to be used on quads with a scale factor less than 0.2. At that scale, the coordinate error is 20%, which makes it's usefulness even more questionable.

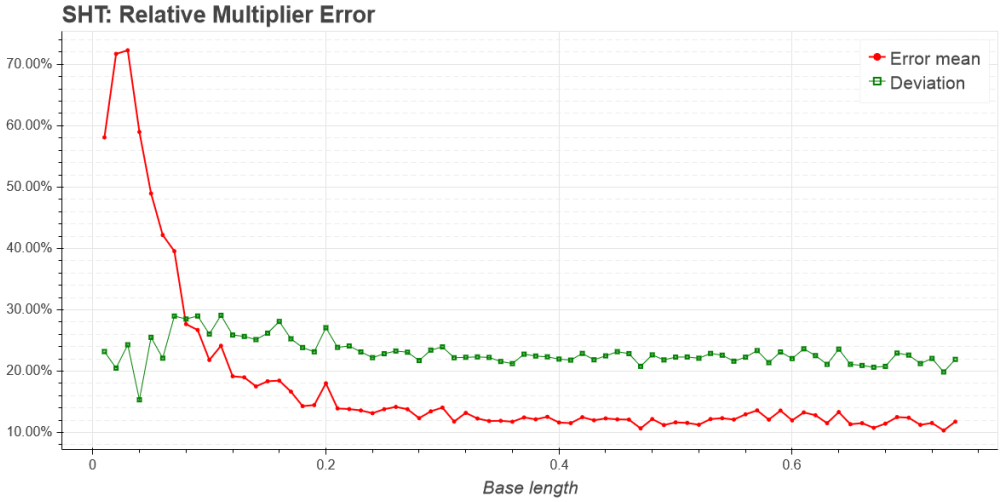

Figure 3.17: *Relative multiplier error with respect to quad size, using the SHT method*

The error is distributed almost evenly between the quad parameters. The base length and the multipliers have on average 15% relative error in their optimal ranges. The angle error

shows a somewhat better picture, with it's 10% minimal error. The deviation is around 20% for all of them. The orientation is also the most accurately measurable parameter with this algorithm. As figure 3.19. shows, the relative error of the orientation stays below 10% for the entire length range. It's minimum is 2%, with 10% deviation.

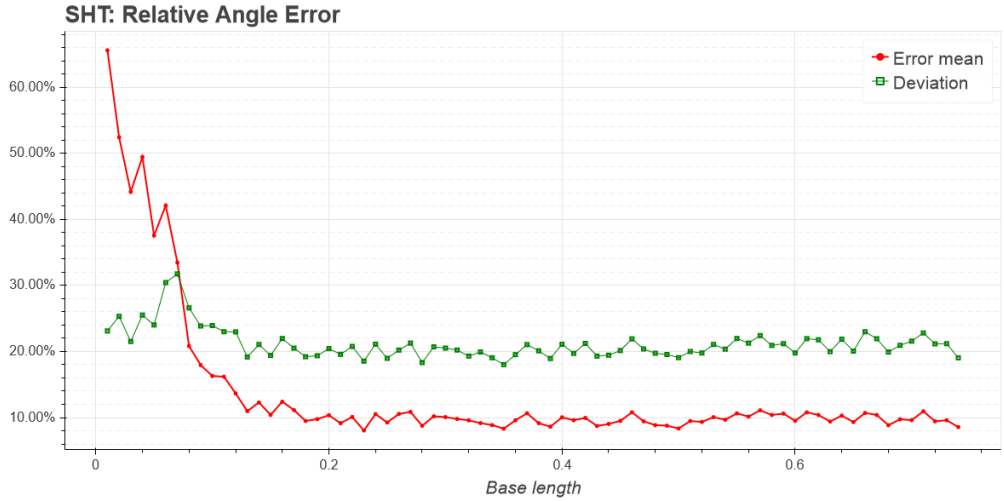

Figure 3.18: *Relative angle error with respect to quad size, using the SHT method*

There are many possibilities to further develop and refine this algorithm to provide much better results. For example, the gradient information could be used to better detect lines, or a Least Squares approximation could be used to fit a line to the pixels marked as inliers to the line. However, all this is true for the algorithm which uses the probabilistic Hough transform, which much better results. Thus, further development of this algorithm is not recommended, nor will it be used further in the project.

Figure 3.19: *Relative orientation error with respect to quad size, using the SHT method*

3.3.5 PHT Quad Detector results

In this section the performance of the probabilistic Hough transform-based quad detector will be evaluated. The inner workings of the detector are very similar to the previously described SHT-based one. Nonetheless, this method is shows both better performance

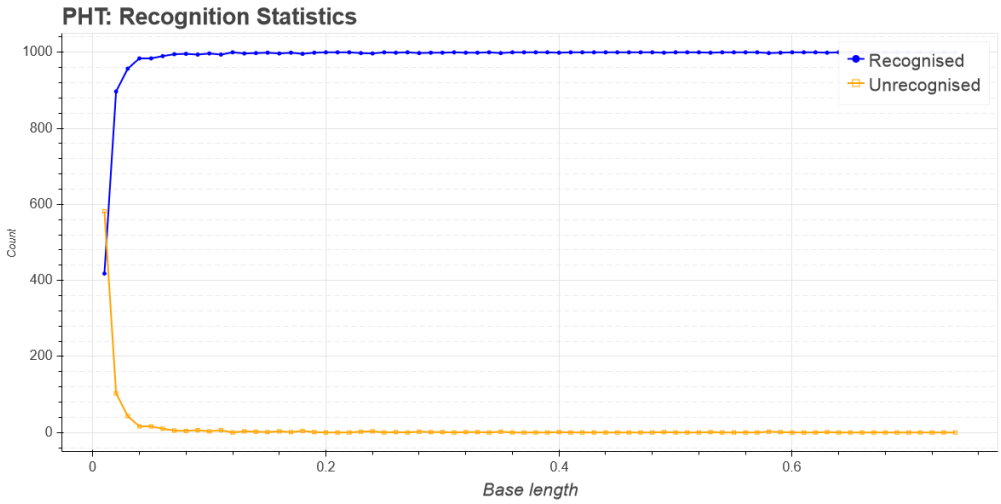

Figure 3.20: *Recognised and unrecognised quads with respect to quad size, using the PHT method*

and requires less computation. It was stated in the theoretical overview of the PHT that reducing the clatter in the accumulator improves the line detection results considerably. Figure 3.20. confirms this. Opposed to SHT, the PHT based method detects even the smallest quads used for experimentation. This detector proved to be the best consistently detecting a quad, although it's accuracy is not unparalleled.

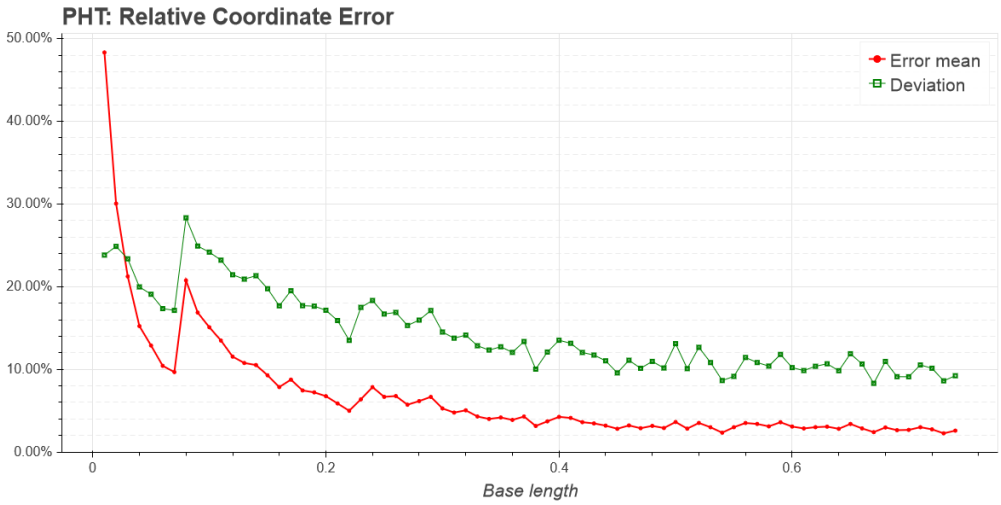

Figure 3.21: *Relative coordinate error with respect to quad size, using the PHT method*

The overall accuracy (that is, average relative error in corner the corner coordinates) is shown in figure 3.21.. Up from 0.1 scale factor the error stays well below 20%, for most sizes

(up from 0.2) below 10%. The minimum error is about 2%. Note that the LSD detector in it's optimal range outperforms this, especially if the deviation is examined.

Figure 3.22: *Relative base length error with respect to quad size, using the PHT method*

The error distribution between the quad parameters follows that of the SHT, only the magnitudes are much smaller. The base length (figure 3.22.), the side multipliers (figure 3.23.), and the angle error (figure 3.24.) are in the same range. For most sizes the error are below 10%, for large quads reaching their minimum of 2%. Their deviation also follows the same trend, declining from 20% to 10%.

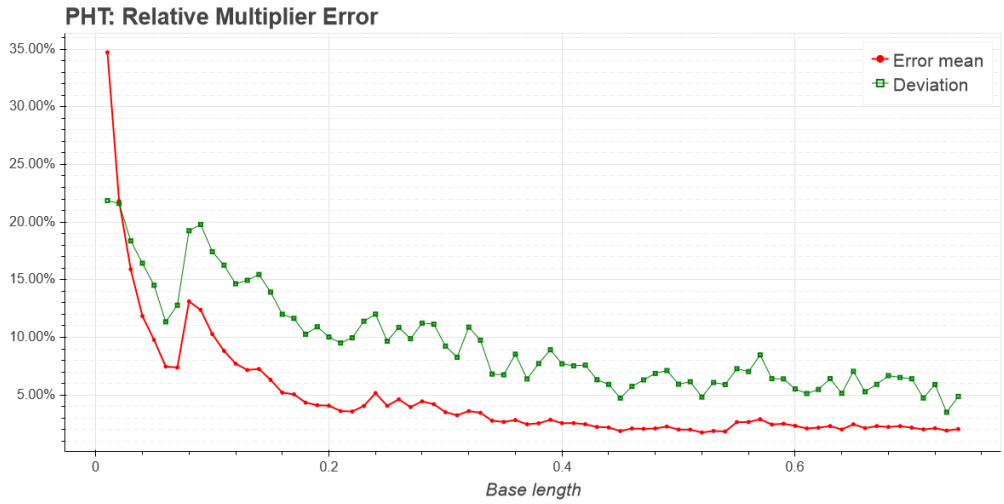

Figure 3.23: *Relative multiplier error with respect to quad size, using the PHT method*

As for the orientation, similarly to the other detectors, it is the most accurately measurable quad parameter. It is always measured within 4% error margin, but for most sizes, within 2%. The reason for this is was explained at the LSD detector: it depends only on the angle of the largest feature of the quad. The error function is shown at figure 3.25..

This method significantly outperforms the SHT quad detector. The two methods share

most of the auxiliary logic between them. They share the heuristics for the Hough parameter calculation (they differ in a scale factor for the threshold, as the PHT works with much less points), the segment matching, and most other functions as well. This significant difference in performance is mainly due to the underlying transformation method.

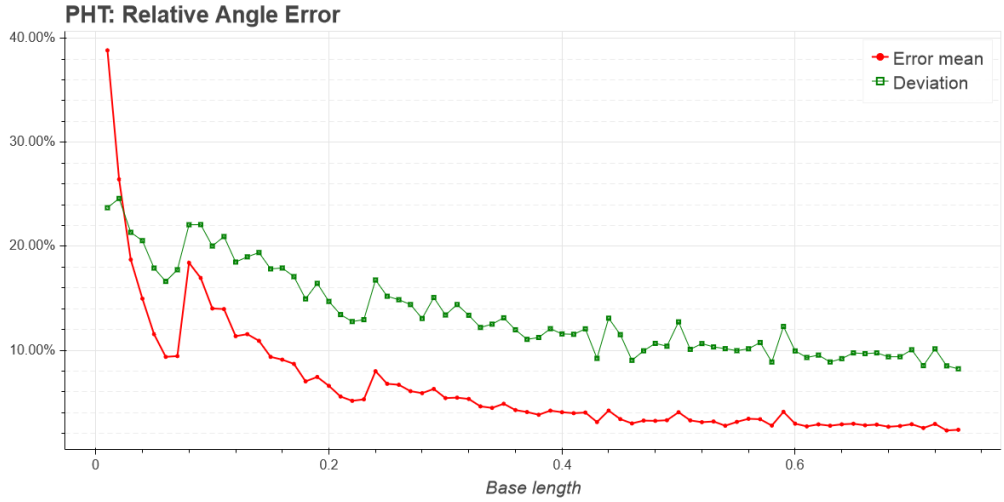

Figure 3.24: *Relative angle error with respect to quad size, using the PHT method*

If we look at the possible refinements of the algorithm, there is plenty. Some have already been mentioned in the previous section, with the SHT algorithm. Better parameter tuning is possible. The skeletoning process which provides the input to the PHT algorithm also can be improved. Incorporating the image gradient information in the transform would also significantly improve the results.

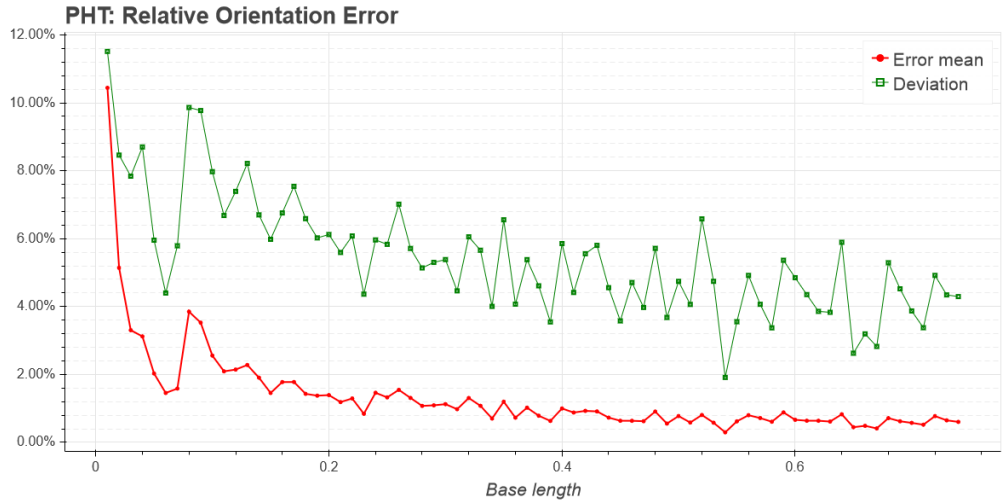

Figure 3.25: *Relative orientation error with respect to quad size, using the PHT method*

3.3.6 Corner Quad Detector results

The final quad detection method, contrary to the so far analysed ones, uses corner detection. It shares the issue of tunable parameters with the Hough transformation based methods. Below is the performance evaluation of the corner detection based quad detector.

As with the rest of the detectors, this one is a prototype, too. This means there are multiple ways to improve it, but it is suitable for some preliminary experiments.

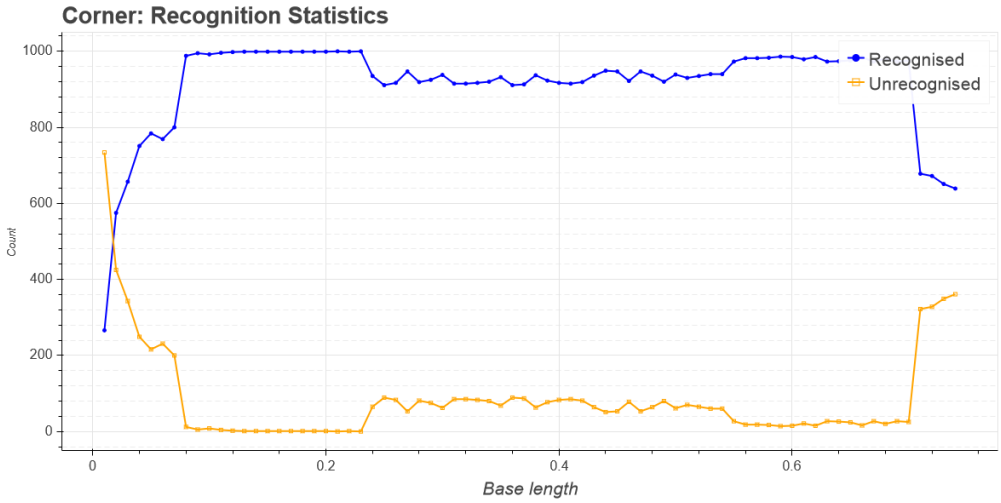

Figure 3.26: *Recognised and unrecognised quads with respect to quad size, using the Corner method*

Figure 3.26. shows the detection statistics of the detector. Aside the really small quads, this method has potential to detect with remarkably good ratio. There are 2 regions where almost all quads were detected. However, in the middle size range, about 10% of the quads were lost. This is due to the rendering process and the lack of optimisation in the detection algorithm. The parameters of the corner detector also need tuning to detect features on

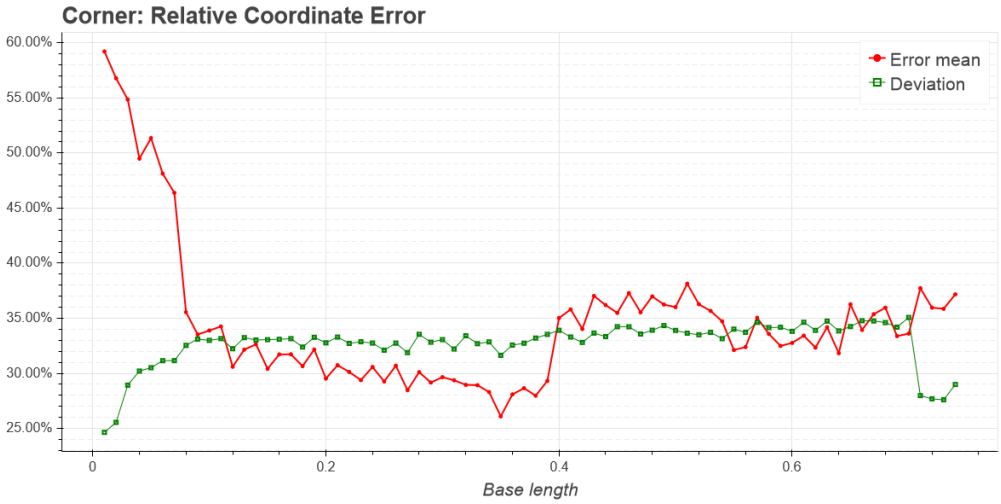

Figure 3.27: *Relative coordinate error with respect to quad size, using the Corner method*

different scales. In the middle region, the corners simply were too "round" to be detected. This can be avoided with scaling of the image, or tweaking the detector parameters.

The same scaling issue can be observed on the error graphs as well. These will not be considered in the error analysis, as they could be eliminated. Only the optimal size range will be analysed.

Figure 3.28: *Relative base length error with respect to quad size, using the Corner method*

The error in the corner positions is shown in figure 3.27.. The magnitude is quite large, comparable to the one produced by the SHT-based method. The reason for this is that the corner detector finds the outer, inner, or both corners os the *edge* of the line, which can be quite far from the centre, depending on the quad angle and the viewpoint. This could be somewhat compensated by using a skeleton, but finding an optimal algorithm for it is problematic.

Figure 3.29: *Relative multiplier error with respect to quad size, using the Corner method*

The error of the base length detection is within reasonable limits, but it is far from being outstanding. In it's optimal region, a 5% error was achieved. The orientation is also detected accurately enough. The multipliers and the angle parameters on the other hand, carry quite a lot of error. Their 30% average value is far from usable, and is outperformed by the other algorithms.

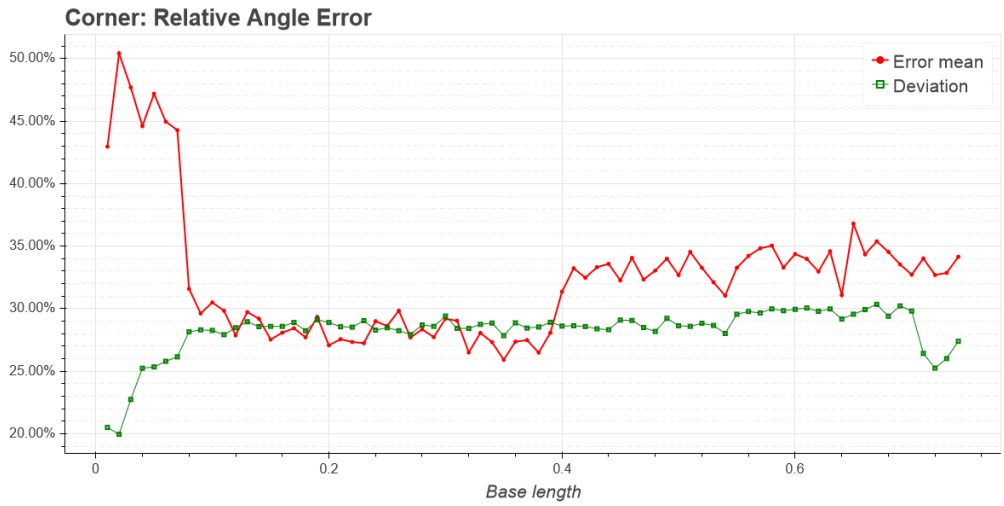

Figure 3.30: *Relative angle error with respect to quad size, using the Corner method*

Based on these results, the application of corner detection is discouraged for this purpose. As for the other algorithms, there are plenty opportunities for improvement, but the other solutions offer better performance overall. Line segment detection seems a more intuitive solution for this problem, and, as expected, produced better results.

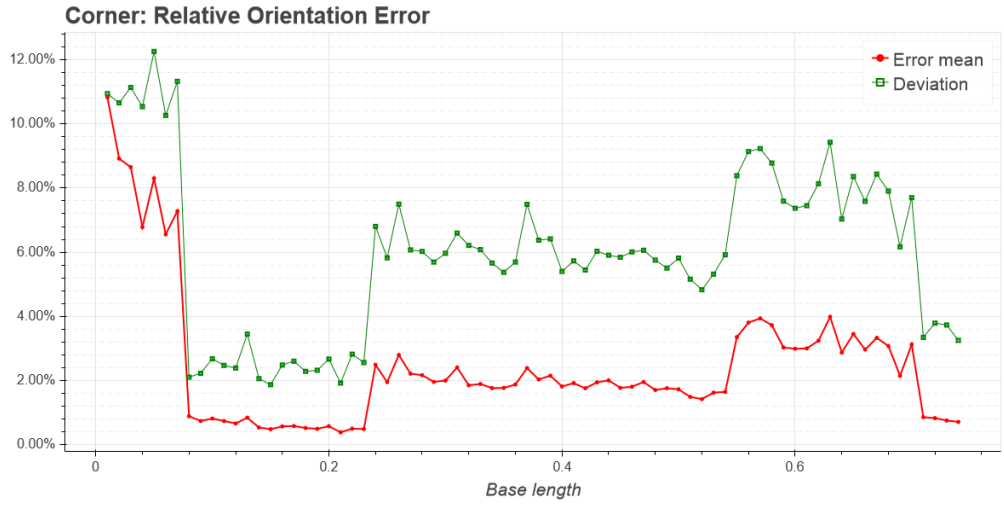

Figure 3.31: *Relative orientation error with respect to quad size, using the Corner method*

3.3.7 Summary

Chapter 4

Marker Recognition

The preprocessing steps used for preparing the images for the line fitters (segmentation, thresholding, filtering etc.) will also be discussed.

The input is the raw image taken[1] by the observer. The first problem is finding the RQIM on the picture. When the marker area is located, it is necessary to discard the only partially visible and/or unrecognisable quads. At this point there is an image or set of images containing potentially good quads.

4.1 Preprocessing

The preprocessing is done in two stages. The first is the segmentation, when quad-like blobs are found. The second step is the preparation of the aforementioned blobs for the line fitting algorithms' needs.

4.2 Segmentation

The segmentation process is carried out on images roughly like the one shown in figure 3.1.. First the photos are converted to binary format by applying a threshold. The image is inverted in the process, because it makes more sense for the objects to be marked with non-zero elements than vice-versa. The threshold's value is determined using Otsu's method, which maximises the inter-class variance of the clusters[2]. The implementation is provided by the OpenCV framework.

Afterwards, the binary image is conditioned with a *close* morphology operator. The closing removes the gaps from the large connected areas (possible quads) and removes the *salt and pepper*-like noise. In the current implementation the kernel size of the morphology operator

[1] In the development phase rendered pictures were used for better repeatability
[2] Foreground and background

is constant, however it could be beneficial to calculate it from the global or local image parameters[3].

The segmentation is based on finding continuous contours on the binary image. The OpenCV framework provides great functionality for this. The implementation is based on calculating the 8-neighbour chain code for the binary blobs on the image. The functions returns a list of list of points for the borders os each distinct contour.

The next step is the filtering of the found blobs. First the surely partial quads are discarded. This is done by calculating the bounding box of the contours, and if one of it's sides are touching the image border, the blob is marked as partial. With this approach it is possible that some fully visible quads that only touch the image border with one of their corner are lost. This problem can be easily fixed by checking the neighbourhood of the contact point, but this is not yet implemented.

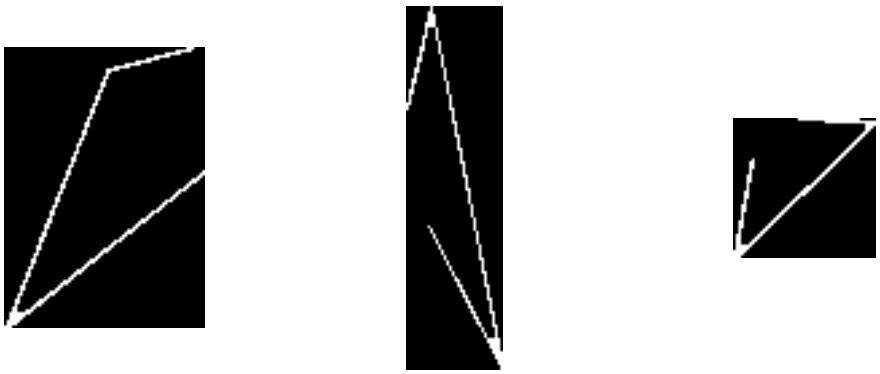

Figure 4.1: *Quad candidates after segmentation*

The next blob-filtering step is to filter out the false-positive contours. These false hits can be caused by the light conditions or the scene around the marker. For this purpose a simple metric is used to measure how likely a blob is to be a quad. This metric is the ratio of a blob's area and circumference. By experimentation this ratio for quads is found to be in the range of 10 and 50. The contours with ratios outside these limits are discarded.

The segmentation processes output is available as a singe image with colour-coded[4] blobs or as a list of separate images each containing a quad candidate. Figure 4.1. shows an example for the output of the segmentation process.

[3] e.g. image size, area of the connected region, etc.
[4] Gray level, to be exact

Chapter 5

Conclusion

Bibliography

[1] D.H. Ballard. Generalizing the hough transform to detect arbitrary shapes. *Pattern Recognition*, 13(2):111 – 122, 1981.

[2] James R Bergen and Haim Shvaytser (Schweitzer). A probabilistic algorithm for computing hough transforms. *Journal of Algorithms*, 12(4):639 – 656, 1991.

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

[4] Rafael Grompone von Gioi, Jeremie Jakubowicz, Jean-Michel Morel, and Gregory Randall. Lsd: A fast line segment detector with a false detection control. 32:722–32, 04 2010.

[5] C. Harris and M. Stephens. A combined corner and edge detector, 1988.

[6] P.V.C. Hough. Method and means for recognizing complex patterns. *U.S. Patent 3,069,654*, Dec. 1962.

[7] N. Kiryati, Y. Eldar, and A.M. Bruckstein. A probabilistic hough transform. *Pattern Recognition*, 24(4):303 – 316, 1991.

[8] J. Matas, C. Galambos, and J. Kittler. Robust detection of lines using the progressive probabilistic hough transform. *Computer Vision and Image Understanding*, 78(1):119 – 137, 2000.

[9] Jianbo Shi and Carlo Tomasi. Good features to track. pages 593–600, 1994.

[10] Lei Xu, Erkki Oja, and Pekka Kultanen. A new curve detection method: Randomized hough transform (rht). *Pattern Recognition Letters*, 11(5):331 – 338, 1990.

Appendix

A.1 Quad Detector Source codes

A.1.1 QuadDetector Base Class

```
class BaseNotFound(Exception):
    pass

class IntersectionNotFound(Exception):
    pass


class QuadDetector:
    """Base class for Quad detectors"""
    def __init__(self):
        self._working_img = None
        self._quad_box_size = None

    def detect_quad(self, img):
        raise NotImplemented("A detector must implement this")

    @staticmethod
    def _find_corners(lines):
        pairs = [pair for pair in itertools.combinations(lines.tolist(), 2)]
        distances = []
        for pair in pairs:
            l1 = shapes.create_line_segment_from_np(pair[0])
            l2 = shapes.create_line_segment_from_np(pair[1])
            distances.append(geom.line_line_distance(l1.a, l1.b, l2.a, l2.b))

        pairs.pop(np.argmax(distances))
        if pairs[0][0] in pairs[1]:
            base = shapes.create_line_segment_from_np(pairs[0][0])
        elif pairs[0][1] in pairs[1]:
```

```
            base = shapes.create_line_segment_from_np(pairs[0][1])
        else:
            raise BaseNotFound

        line_list = [shapes.create_line_segment_from_np(line) \
                     for line in lines.tolist()]
        line_list.remove(base)

        inner = []
        outer = []
        for line in line_list:
            A = ((line.b.y - line.a.y, line.a.x - line.b.x),
                 (base.b.y - base.a.y, base.a.x - base.b.x))
            B = (line.a.x * line.b.y - line.b.x * line.a.y,
                 base.a.x * base.b.y - base.b.x * base.a.y)
            A = np.stack(A)
            try:
                intersect = np.linalg.solve(A,B)
            except np.linalg.LinAlgError:
                raise IntersectionNotFound

            dist = [geom.distance(intersect, point) \
                    for point in line.get_endpoints()]
            inner.append(shapes.Point2D(intersect[0], intersect[1]))
            outer.append(line[np.argmax(dist)])

        return [outer[0], inner[0], inner[1], outer[1]]

    def _set_bounding_box_size(self):
        contours = cv2.findContours(self._working_img, cv2.RETR_EXTERNAL,
                                                       cv2.CHAIN_APPROX_NONE)
        min_rect = cv2.minAreaRect(contours[1][0])
        box = cv2.boxPoints(min_rect)
        dist = [geom.distance(box[0], pt) for pt in box[1:]]
        dist.sort()
        self._quad_box_size = tuple(dist[0:-1])
```

A.1.2 LSDQuadDetector Class

```
class LSDQuadDetector(QuadDetector):
    def __init__(self):
        super().__init__()
```

```
        self.lsd = cv2.createLineSegmentDetector()

    def _find_parallel(self, point, other_points):
        distances = [geom.distance(point, -other) for other in other_points]
        return np.argmin(distances)

    def _find_pairs(self, lines):
        dir_vectors = []
        for line in lines:
            dir_vector = line[2:4] - line[0:2]
            dir_vectors.append(dir_vector / np.linalg.norm(dir_vector))

        pairs = []
        for i in range(len(dir_vectors)):
            temp_vectors = dir_vectors.copy()
            min_idx = self._find_parallel(dir_vectors[i], temp_vectors)
            idx_pair = [min_idx, i]
            idx_pair.sort()
            pairs.append((lines[idx_pair[0]], lines[idx_pair[1]]))

        pairs = np.stack(pairs)
        pairs = np.unique(pairs, axis=0)
        return pairs

    def _merge_pairs_long(self, pairs):
        lines = []
        for pair in pairs:
            l1 = shapes.create_line_segment_from_np(pair[0], pair[0][4])
            l2 = shapes.create_line_segment_from_np(pair[1], pair[1][4])
            if l1.get_length() > l2.get_length():
                norm = np.concatenate((l1.get_norm_vector(),
                                       l1.get_norm_vector(), [0]))
                norm = norm * (l1.width / 2)
                lines.append(pair[0] + norm)
            else:
                norm = np.concatenate((l2.get_norm_vector(),
                                       l2.get_norm_vector(), [0]))
                norm = norm * (l2.width / 2)
                lines.append(pair[1] + norm)

        return np.stack(lines)
```

```
    def _scale_to_quad_space(self, corners):
        img_size = self.working_img.shape
        return [corner.scale_inhomogen(1/img_size[0], 1/img_size[1])
                       .translate((-0.5, -0.5)) for corner in corners]

    def detect_quad(self, img):
        self.working_img = img
        try:
            lines, widths, prec, nfa = self.lsd.detect(img)
            lines = [(np.concatenate((lines[i][0], widths[i]))) \
                      for i in range(lines.shape[0])]
            lines = np.stack(lines)
        except AttributeError:
            return None

        # Both edges of every line segment is found
        if lines.shape[0] == 6:
            pairs = self._find_pairs(lines)
            lines_merged = self._merge_pairs_long(pairs)
        elif lines.shape[0] == 3:
            lines_merged = lines
        else:
            return None

        try:
            corners = self._find_corners(lines_merged)
        except (BaseNotFound, IntersectionNotFound) as e:
            return None

        if corners:
            return quad.Quad(self._scale_to_quad_space(corners))
        return None
```

A.1.3 HoughQuadDetector Class

```
class HoughQuadDetector(detector.QuadDetector):
    def __init__(self):
        super().__init__()
        self._skeleton = None

    def _init_iteration(self, img):
        self._orig_img = img
```

```
        if img.shape[2] == 3:
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        self._working_img = cv2.threshold(img, thresh=127, maxval=255,
                                          type=cv2.THRESH_BINARY_INV)[1]
        self._set_bounding_box_size()
        self._skeletonize()

    def _skeletonize(self):
        self._skeleton = np.zeros(self._working_img.shape, np.uint8)
        element = cv2.getStructuringElement(cv2.MORPH_CROSS, (3, 3))

        done = False
        img = np.copy(self._working_img)
        while not done:
            eroded = cv2.erode(img, element)
            temp = cv2.dilate(eroded, element)
            temp = cv2.subtract(img, temp)
            self._skeleton = cv2.bitwise_or(self._skeleton, temp)
            img = np.copy(eroded)
            done = cv2.countNonZero(img) == 0

    def _scale_to_quad_space(self, corners):
        img_size = self._working_img.shape
        return [corner.scale_inhomogen(1/img_size[0], 1/img_size[1])
                      .translate((-0.5, -0.5)) for corner in corners]

    def detect_quad(self, img):
        self._init_iteration(img)

        try:
            segments = self._find_segments()
        except NoLinesDetected:
            return None

        line_segments = []
        for seg in segments:
            line_segments.append(np.concatenate(seg.get_endpoints()))

        if len(line_segments) < 3:
            return None
```

```
        line_segments = np.stack(line_segments)
        try:
            corners = self._find_corners(line_segments)
        except (BaseNotFound, IntersectionNotFound):
            return None

        return quad.Quad(self._scale_to_quad_space(corners))
```

ClassicalHoughQuadDetector Class

```
class ClassicHoughDetector(HoughQuadDetector):
    def __init__(self):
        HoughQuadDetector.__init__(self)

    def _get_threshold(self):
        min_dist = np.min(self._quad_box_size)
        if min_dist < 15:
            return 5
        return int(min_dist / 3)

    def _find_lines(self):
        thresh = self._get_threshold()

        lines = cv2.HoughLines(self._skeleton, 1, np.pi / 180, thresh)
        if lines is None or len(lines) < 3:
            raise NoLinesDetected
        lines = [line[0] for line in lines]
        lines = np.stack(lines)

        criteria=(cv2.TERM_CRITERIA_EPS+cv2.TERM_CRITERIA_MAX_ITER,10,0)
        compactness, labels, centers = cv2.kmeans(lines, 3, None,
                                            criteria=criteria,
                                            attempts=10,
                                            flags=cv2.KMEANS_RANDOM_CENTERS)

        return centers

    def _find_segments(self):
        lines = self._find_lines()

        segments = []
        for rho, theta in lines:
```

```
            cos = np.cos(theta)
            sin = np.sin(theta)
            if np.isclose(sin, 0):
                line_points = ((y, int(((rho - y * sin) / cos))) \
                               for y in range(self._working_img.shape[0]))
            else:
                line_points = ((int((rho - x * cos) / sin), x) for x in \
                               range(self._working_img.shape[1]))

            try:
                max_line_gap = int(min(self._quad_box_size) / 2)
                line_points = (pt for pt in line_points \
                               if is_valid_index(pt, self._working_img.shape))
                segment_points = [(pt[1], pt[0]) for pt in line_points \
                                  if self._working_img[pt]]
                segments.append(geom.LineSegment2D(segment_points[0],
                                                   segment_points[-1], 0))
            except IndexError:
                raise NoLinesDetected

        return segments
```

ProbabilisticHoughQuadDetector Class

```
class ProbabilisticHoughDetector(HoughQuadDetector):
    def __init__(self):
        HoughQuadDetector.__init__(self)

    def _get_threshold(self):
        min_dist = np.min(self._quad_box_size)
        return int(max((5, min_dist / 6)))

    def _merge_line_segments(self, segments, dir_vec):
        points = [seg.a for seg in segments]
        points.extend([seg.b for seg in segments])

        base = points.pop()
        line_vectors = [point - base for point in points]
        skalar_produkts = [np.dot(dir_vec, line_vec) \
                           for line_vec in line_vectors]

        min_value = min(skalar_produkts)
```

```
        max_value = max(skalar_produkts)
        min_idx = np.argmin(skalar_produkts)
        max_idx = np.argmax(skalar_produkts)

        if min_value < 0 and max_value < 0:
            return geom.LineSegment2D(points[min_idx], base, 0)

        if min_value > 0 and max_value > 0:
            return geom.LineSegment2D(points[max_idx], base, 0)

        return geom.LineSegment2D(points[min_idx], points[max_idx], 0)

    def _merge_segments(self, segments):
        dir_vectors = [seg.get_dir_vector() for seg in segments]
        dir_vectors = np.array(dir_vectors, dtype=np.float32)
        if np.isnan(dir_vectors).any():
            raise NoLinesDetected

        criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 0)
        compactness, labels, centers = cv2.kmeans(dir_vectors, None,
                                                  criteria=criteria,
                                                  attempts=10,
                                                  flags=cv2.KMEANS_RANDOM_CENTERS)

        segment_clusters = [[],[],[]]
        for label, segment in zip(labels.tolist(), segments):
            segment_clusters[label[0]].append(segment)

        merged_segments = []
        for dir_vec, lines in zip(centers, segment_clusters):
            if len(lines) > 1:
                merged_segments.append(self._merge_line_segments(lines, dir_vec))
            else:
                merged_segments.append(lines[0])

        return merged_segments

    def _find_segments(self):
        thresh = self._get_threshold()
        min_line_length = min(self._quad_box_size) / 4

        lines = cv2.HoughLinesP(self._working_img, 1, np.pi / 180,
```

```
                                thresh, minLineLength=min_line_length,
                                maxLineGap=min_line_length/2)
        if lines is None or len(lines) < 3:
            raise NoLinesDetected

        line_segments = [geom.create_line_segment_from_np(seg[0]) \
                         for seg in lines]
        line_segments = self._merge_segments(line_segments)

        return line_segments
```

A.1.4 CornerQuadDetector Class

```
class CornerQuadDetector(QuadDetector):
    def __init__(self):
        super().__init__()

    def _init_iteration(self, img):
        self._orig_img = img
        if img.shape[2] == 3:
            img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        self._working_img = cv2.threshold(img, thresh=127,
                                          maxval=255,
                                          type=cv2.THRESH_BINARY_INV)[1]
        self._set_bounding_box_size()

    def _identify_points(self, centroids):
        binary = cv2.threshold(self._working_img,
                               thresh=127, maxval=255,
                               type=cv2.THRESH_BINARY_INV)[1]

        pointpairs = itertools.combinations(centroids.tolist(), 2)
        lines = []
        for pointpair in pointpairs:
            line_points = geom.createLineIterator(np.array(pointpair[0],
                                                  dtype=np.int_),
                                                  np.array(pointpair[1],
                                                  dtype=np.int_),
                                                  binary)
            hit_miss_ratio = np.count_nonzero(line_points[:,2]) /
                             len(line_points)
            lines.append((hit_miss_ratio, pointpair))
```

```
        lines.sort(key=lambda line: line[0])
        lines = lines[0:3]
        lines = [line[1:][0] for line in lines]

        line_points = [point for line in lines for point in line]
        outer = []
        inner = []
        for centroid in centroids.tolist():
            point_occurance_count = line_points.count(centroid)
            if point_occurance_count == 1:
                outer.append(centroid)
            elif point_occurance_count == 2:
                inner.append(centroid)
            else:
                raise BaseNotFound

        return inner, outer

    def _scale_to_quad_space(self, corners):
        img_size = self._working_img.shape
        return [corner.scale_inhomogen(1/img_size[0], 1/img_size[1])
                       .translate((-0.5, -0.5)) for corner in corners]


    def _merge_corners(self, corners):
        if len(corners) == 4:
            return corners

        corner_list = [corner[0] for corner in corners.tolist()]
        pairs = itertools.combinations(corner_list, 2)
        pairs = [(pair, geom.distance(pair[0], pair[1])) \
                 for pair in pairs]
        pairs.sort(key=lambda el: el[1])
        closest = pairs[0][0]

        corner_list.remove(closest[0])
        corner_list.remove(closest[1])
        corner_list.append([(closest[0][0] + closest[1][0]) / 2,
                            (closest[0][1] + closest[1][1]) / 2])

        corner_list = [[corner] for corner in corner_list]
```

```
        return self._merge_corners(np.array(corner_list))

    def detect_quad(self, img):
        self._init_iteration(img)

        gray = np.float32(self._working_img)
        corners_good = cv2.goodFeaturesToTrack(gray, 6, 0.1, 2)
        if len(corners_good) < 4:
            return None
        corners_good = self._merge_corners(corners_good)
        corners_good = np.stack([corner[0] for corner in corners_good])

        try:
            inner, outer = self._identify_points(corners_good)
        except (BaseNotFound, TypeError) as e:
            return None

        corners = [outer[0], inner[0], inner[1], outer[1]]
        corners = [bg.Point2D(point[0], point[1]) for point in corners]
        return quad.Quad(self._scale_to_quad_space(corners))
```