Universität Osnabrück

Human science faculty, Institute of Cognitive Science

In cooperation with the eyeTrax® project, mindQ GmbH & Co. KG (Osnabrück)

# Recreating a fully automatic eye-tracking calibration algorithm based on geometric reprojection with a subsequent analysis with noisy data

*A THESIS SUBMITTED FOR THE DEGREE OF COGNITIVE SCIENCE (B.Sc.)*

**Author: Marko Duda**
Matriculation number: 962763

**Date of submission: 07.06.2019**

Supervisor: Dr. Matthias Temmen // Prof. Dr. Peter König

*Abstract*

Neurological diseases like concussions lead to changes in the oculomotor system. eyeTrax, a new digital diagnosis technology, uses eye tracking to tests patients' oculomotor system in virtual reality. It is designed to provide a diagnosis for concussions within minutes. Typical for digital diagnosis tools, it is an ambulant technology and it is cost-effective. eyeTrax won the 2nd price of the Novartis digital health award 2019. In the future, eyeTrax might be able to detect other neurological diseases like ADHD or dementia as well.

In this thesis, the self-calibrating eye tracking algorithm, eyeTrax uses, will be implemented in Python. No other codebase, only a conceptual description, was used as a draft. It is the first Python-only implementation of this algorithm. Today's eye tracking systems are designed to deliver real time results. They do this at the expense of tracking accuracy. Since eyeTrax is a medical application, accuracy and reliability are more important than a good runtime. Python is known for high-readability and fast prototyping. This makes this thesis implementation the ideal starting point for an eye-tracking software with its focus on high accuracy and reliability.

Furthermore, the robustness of this thesis implementation is tested with noisy data and compared to the ground truth data.

# Table of Content



**Kommentiert [MD1]:** Aufteilung
Intro method result discussion
20 max 20 min 40 min 20 max

# 1 Introduction

Digital diagnostics is one of the big trends in the sector of digital health. The benefits are obvious: low costs, scalability, fast diagnosis and portability. Digital diagnostic is simple in its application. Neurological diseases like craniocerebral trauma, strokes, ADHD, dementia and Parkinson lead to changes in the oculomotor system (Antoniades et al., 2014). With an aging society, especially neurodegenerative diseases like Parkinson and dementia are on a rise.

*Figure 1 eyeTrax logo*

Nowadays, doctors test patient's oculomotor system manually. Gross evaluation of conjugate eye movements is commonly performed by doctors. "Disconjugate eye movements have been associated with traumatic brain injury since ancient times." (Altomare et al., 2015, p.548) However, with the help of modern eye tracking technology, neurological diseases are available for digital diagnostic.

This thesis was made in cooperation with the company mindQ. They develop eyeTrax. eyeTrax is a new digital diagnosis technology to diagnose neurological diseases. With eyeTrax, patients do visual tests in virtual reality. A visual test for example can be that patients have to simply follow a point with their eyes. eyeTrax than uses eye tracking to detect anomalies in the oculomotor system.

Current eye tracking software as well as the one that eye tracking uses is optimized for real time use. This fast running time comes at the cost of tracking accuracy. If not executed in real time, eye tracking accuracy could be better. For medical application, high accuracy is especially important. In this thesis, the eye tracking method eyeTrax uses will be rebuild in Python. Python has not the best running time but it is the ideal language in terms of readability, fast prototyping. Furthermore, it is the programming language of machine learning and data science.[1] Making this thesis implementation, the ideal starting point for a high-accuracy eye tracking software.

## 1.1 eyeTrax

As already mentioned, eyeTrax is a new technology to diagnose neurological diseases. It let patients do tests with visual stimuli in virtual reality and uses eye tracking to detect anomalies in the oculomotor system. It won the second price of the Novartis digital health award 2019 which is notated with 15.000€.

Its first use case will be the diagnosis of concussions. After a concussion, patients often are not able to fixate a visual stimulus directly like healthy people do. They overshoot the stimuli and need to adjust the stimuli with a second eye fixation (see "Einsatz bei der AIBA Box WM 2017 Hamburg" at eyetrax.de). eyeTrax is an ambulant technology. This makes it practical for certain top-class sports. Boxers, rugby players and footballers for example are highly endangered of suffering from traumatic brain injuries. In the German Bundesliga, one concussion happens every second game day (see "Kopfverletzungen im Fußball – Dr.Helge Riepenhof im NDR Sportclub" at eyetrax.de). The world football federation FIFA permitted 3 minutes break during a game to detect a possible concussion of a player.

[1] Machine learning and data science methods could be used to improve the eye tracking accuracy, too

This is just enough time for the eyeTrax diagnosis. Furthermore, VR has the benefit that the patient is isolated from disturbing external stimuli during the examination.

Light concussions need no special treatment. The patient just has to rest. A second concussion, however, within a short time period can be really dangerous. Long-term symptoms can be light depression, headache, dementia and similar conditions. There were even cases of American football players who died after suffering from repeated concussions during a game.


*Figure 2 Patients have to find the highest number from the rotating dices*

How does eyeTrax work? State-of-the-art high-performance eye tracking cameras are built in the virtual reality glasses of the eyeTrax system. Manual calibration of the eye tracking system needs the patient to fixate a visual stimulus – a cross for example. Patients who suffered from neurological diseases might not be able to fixate a visual stimulus properly. So, there is no manual calibration. eyeTrax calibrates itself automatically. Patients than have to do tasks in VR that test their oculomotor system. Figure 2 shows one example of such a task. These tasks are designed to reproducibly trigger eye movements and pupil reactions relevant for diagnostic examination.

The tasks result can be compared intraindividually (a boxer can be tested with eyeTrax before and after multiple head hits). If not interandividual, a norm variant can be used to compare to. This norm variant is a cumulation of various measurement of different persons.

eyeTrax uses relative eye movements for diagnosis which means that it is only measured how much the eye is moving to a certain direction. There is no gaze-to-world mapping. Contrary to comparable systems, eyeTrax is not limited to the analysis of individual oculomotor functions but can take all measured data like biomarkers comprehensively into account. In order to do so, it uses the latest machine learning and data science methods like deep neural networks or support vector machines.

## 1.2 Eye tracking calibration methods

This subsection gives a short overview about different concepts and methods used for eye tracking calibrations. Then, the advantage of the method eyeTrax uses will be explained.

All eye tracking methods have in common that they need to be calibrated in some form. Otherwise, they could not map a pupil position to a gaze. "Most gaze estimation algorithms can be classified into two groups: regression-based and model-based. Regression-based algorithms assume that there is some unknown relationship between the detected eye parameters and the gaze. They then approximate this relationship using some form of regression [...]. Model-based approaches instead attempt to model the eye and thus the gaze and [..] output the gaze information." (Swirski, 2013, p.1)

Both approaches require usually some form of manual calibration. Subjects have to fixate crosses for calibration for example. As already mentioned, eyeTrax is able to calibrate itself automatically. It does so with the method of the paper "A fully-automatic, temporal approach to single camera, glint-free 3D eye model" (Swirski, 2013). From now on this method is referred to as the Swirski-algorithm. This method works with the projective geometry of the pupil. The pupil is detected as an ellipse in the image frame. Analytical geometry is then used to approximate the orientation and position of the original 3D pupil contour circle. By combining the information of these 3D circles frames together, the eyeball's position is estimated and then the eye model is built.

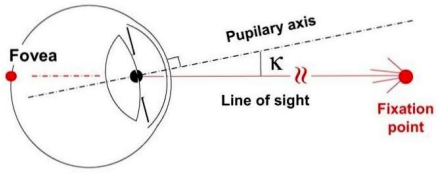



*Figure 3 Offset K between pupillary axis and line of sight*

The offset K between the pupilary axis and the line of sight (see Figure 3) is not consider by Swirski. Only the pupilary axis will be tracked. Offset K is constant for an individual. For relative eye movements (as used by eyeTrax) this offset does not make a difference, since the relative motion stays the same. From now on and for the matter of simplicity, the pupillary axis will be referred to as the gaze.

## 1.3 Why developing an own implementation from scratch?

The aim of this thesis is to build an accuracy focused implementation of the Swirski-algorithm. Only runtime focused versions exist so far. How exactly differs an accuracy focused version of this algorithm from a runtime focused version?

In order to deliver real time results, the Swirski-algorithm has to process the pupil frames in small serial batches. If it wants to use all pupil frames at once, it would have to wait towards the end of the tracking session until all frames will be recorded. Of course, approaches using and process the frames of the whole session at once are more accurate. Singular outliers does not matter as much they would in a small batch.

The question that now arises is whether existing implementations of the Swirski-algorithm can be easily changed to be accuracy focused. To answer this question, first, some more technical background knowledge about the topic itself is necessary: the Swirski-algorithm uses a 3D eye model. So thematically, the Swirski-algorithm lays in the field of 3D computer graphics and works with a coordinate system.[2]

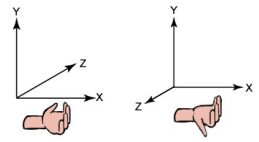



*Figure 4 Left and right handed coordinate system*

There are a lot of configurational questions need to be answered for working with a coordinate system in 3D computer graphics: Which kinds of coordinate systems is used: right and left handed (see Figure 4)? In which direction inside the coordinate system shows the camera? Where is the camera positioned? Where is the image plane inside the coordinate system?

It is almost impossible to answer these configurational questions by the source code alone. The source code's documentation has to answer it. Currently, there are two open-source implementations of the Swirski-algorithm available. The one of Swirski himself and the one of Pupil Labs, a company from Berlin, selling eye tracking systems. eyeTrax uses the pupil labs hardware as well as some of its software, too.

Unfortunately, both implementations are not documented so well. It is not really possible to answer the configurational questions stated above from the documentation. Furthermore, they consist of thousands of lines of code. So, it is easy to lose the overview. Both use C++. Pupil Labs additionally uses some Python for its implementation. In order to let both programming languages work together, Cython, a bridge between Python and C++ is used.

Because of all of these aspects, it was decided to program the Swirski-algorithm from scratch. Only the conceptual description of the paper is used as a draft. Of course, this approach has some disadvantages, too. A conceptual draft is very often not enough for an algorithmic understanding. In order

[2] It does not work with polygons as it is usual in computer graphics. So thematically, it is still very special inside this field but the coordinate sy

to get towards an algorithmic understanding, own assumptions and conclusion have to be made. Nonetheless, it is believed that this approach will be the better one.

## 1.4 Terminology

Since the subject of this thesis is rather technical, a clear terminology helps to not confuse things. This short terminology should be looked up if needed.

| | |
|---|---|
| **3D circle** | 3D representation of pupil contour |
| **2D ellipse** | Representation of pupil contour projected to 2D and skewed by perspective projection to an ellipse |
| **pupil** | The pupil contour either represented in 2D or 3D |
| **image plane** | The plane in 3D space on which the 2D ellipse rest on |
| **implicit** | An implicit equation equals zero. Implicit equations are one form of representing geometric figures mathematically. For example, $x^2 + y^2 - 1 = 0$ is the implicit form of a circle. |
| **parametric** | Another form to mathematically represent a geometrical figure. Contrary to the implicit form, it needs an additional variable $\theta$. The x coordinate of a circle is represented with $x = r * cos(\theta) + x_{Center}\ with\ \theta \in \{0, 2\pi\}$ |
| **input data** | The set of 2D elliptic pupil contours gained during an eye tracking session |

# 2 Concepts

This section is about the main conceptual frameworks, that were used to implement the eye tracking algorithm later on.

## 2.1 Projection Simulation

This subsection is about the simulation of an eye tracking session. Why is it necessary to do that? The input data consist of the two-dimensional elliptic pupil contours recorded during an eye tracking session. By simulating the input, ground truth data is available and can compared with the data gained by the eye tracking algorithm. This is not only important for further robustness analysis where noise is added to this input data to see how the eye tracking algorithm behaves under uncertainty. It is in particular important for software testing as well. Without ground truth data, one would not know how well the algorithm actually performs.

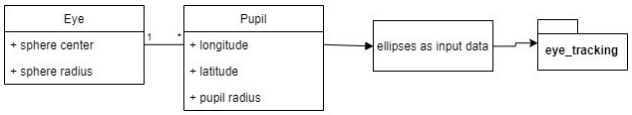



*Figure 5 UML of generating the input data*

Above you can see an UML of how the input data is generated. The eye is described by the coordinates of its center and its radius. The coordinate system origin is the focal point of the camera. The pupil is described by its longitude, latitude and the radius of the pupil. Figure 6 shows an example for the input data.

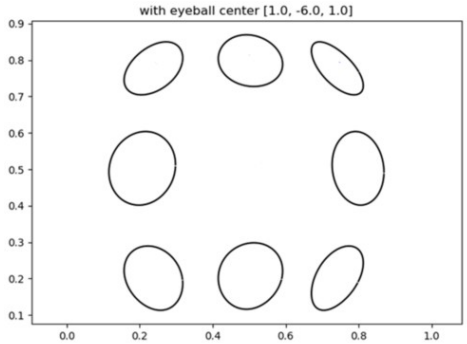



*Figure 6 Example input data for the eye tracking algorithm*

## 2.2 Principal Axis Theorem (Background)

To understand the mathematics of the eye tracking algorithm some background knowledge about quadrics is necessary beforehand. Ellipsis are one kind of quadrics. A quadric is a zero set of a quadratic multivariable polynomial equation. Its general implicit form for two dimensions is

$$ax^2 + bxy + cy^2 + dx + ey + f = 0 \qquad (1)$$

Quadrics can form geometric figures. For example, the zero set of the following equation

$$\left(\frac{x}{2.49}\right)^2 + \left(\frac{y}{3.22}\right)^2 = 1 \qquad (2)$$

forms an ellipse. It has the semi axis length of 3.22 and 2.49[3]. Its properties are easily observable in the denominator of (2). Shifting this ellipse so that its center is the point (-1.53|1.13) and rotate it 45° counter-clockwise gives the following equation.

$$4x^2 + 2xy + 4y^2 + 10x - 6y - 20 = 0 \qquad (3)$$

By looking at the coefficients alone, (3) is not interpretable as an ellipse with the mentioned properties. The so-called principal axis theorem (PAT) transforms the ellipse back to its normal form (2) so that its properties become visible again. The PAT consists of first rotating and then shifting the quadric such that its major and minor axis rest on the coordinate axis again (see Figure 7).

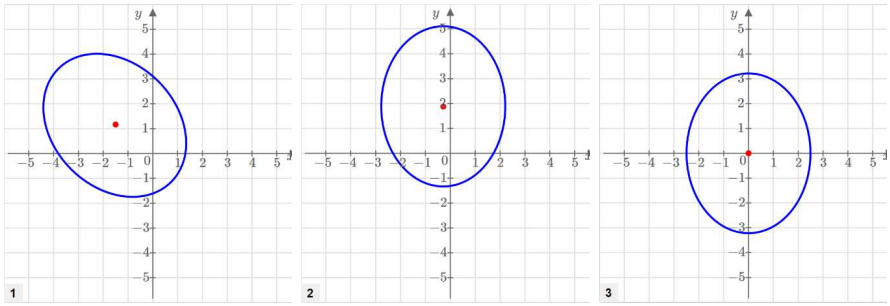



*Figure 7 Principal axis theorem applied to an ellipse*

### 2.2.1 Rotation

A rotation is nothing else than a linear transformation. A coordinate is transformed from point *(x|y)* of Figure 7.1 to the point *(x'|y')* of Figure 7.2. After the rotation, the mixed term *bxy* disappears from (3). To prepare the rotation, the quadric needs to be represented in its matrix form (4), which is just another form of (1).

$$(x \quad y)\begin{pmatrix} a & \frac{b}{2} \\ \frac{b}{2} & c \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + (d \quad e)\begin{pmatrix} x \\ y \end{pmatrix} + f = 0 \qquad (4)$$

By diagonalizing $B = \begin{pmatrix} a & \frac{b}{2} \\ \frac{b}{2} & c \end{pmatrix}$, both of its elements $\frac{b}{2}$ are set to zero. Matrix *B* is symmetric and therefore an additional check whether it is diagonalizable is not necessary. The diagonalization of *B* is matrix *D* with the eigenvalues of B which are $\lambda_1$ and $\lambda_2$ as its elements.

$$D = \begin{pmatrix} \lambda 1 & 0 \\ 0 & \lambda 2 \end{pmatrix} \qquad (5)$$

Next step is to get the transformation matrix *T* for which the following condition holds place:

[3] The properties of (2) and (3) are rounded for didactic purposes.

Kommentiert [MD2]: ((((x-1)cos(A))+y-ksin(A))^(2))/(a^(2))+(((x-h)sin(A)-(y-k)cos(A))^(2))/(b^(2))=1

Kommentiert [MD3]: Rotation does not mean that one axis lays on the coordinate axis. Since then there would not be both non-quadric terms involved

$$\binom{x}{y} = T\binom{x'}{y'} \qquad (6)$$

The linear transformation $T$ describes a rotation. For congruency of the ellipse, $T$ needs to be an orthogonal matrix which means that its row and column vectors are orthonormal.[4] The columns of $T$ consist of the eigenvectors of $B$, they have the same order as the eigenvalues in $D$. The resulting cone is calculated with the following formula

$$(x' \quad y')D\binom{x'}{y'} + (d \quad e)T\binom{x'}{y'} + f = 0 \qquad (7)$$

and for the equation (3)

$$5x'^2 + 3y'^2 + 2\sqrt{2}x' - 8\sqrt{2}y' - 20 = 0 \qquad (8)$$

### 2.2.2 Shifting

For the shifting, the non-quadric terms *dx* and *ey* from (8) need to disappear. Point *(x'|y')* from Figure 7.2 get transformed to point *(X|Y)* from Figure 7.3. For the shifting, the method of completing the square is applied for both the *x'* as well as the *y'* term. The exact procedure is not important for this thesis and will therefore only shortly explained. Applying the completing the square method for (8) results in

$$5\left(x' + \frac{\sqrt{2}}{5}\right)^2 + 3\left(y' - \frac{4\sqrt{2}}{3}\right)^2 - \frac{466}{15} = 0 \qquad (9)$$

By (9) the center of the ellipse of Figure 7.2 can be read which are $\left(-\frac{\sqrt{2}}{5} \middle| \frac{4\sqrt{2}}{3}\right)$ by replacing the braces by *X* and *Y*, one gets

$$5X^2 + 3Y^2 - \frac{466}{15} = 0 \qquad (10)$$

which is the same as equation (2).

## 2.3 Ellipse reprojection

The math and the conceptual framework of the geometric reprojection is basically described by two papers. The first one, "Three-dimensional location estimation of circular feature" by Safaee-Rad et al. (Safaee-Rad et al., 1992), is about the math on how to find a 3D circle given its projection by a 2D ellipse (see Figure 8). Quadrics and PAT are important for this. As can be seen, there are two possible 3D solutions: the red and green 3D circle, that is the so called **two-circle ambiguity**. Each circle has its own **orientation**, represented by the red and green arrow but both share the same circle **position** which is represented by the circle's center position.[5]

Safaee-Rad takes the 2D ellipse as an input, calculates the cone from it and uses this cone to derive the position and orientation of the 3D circle. This section will describe how all of this is done.

---

[4] The transpose of an orthogonal matrix is its inverse at the same time. This means that $U^T U = E$ which makes software unit tests trivial for this case.

[5] The method can be easily expanded to spheres instead of 3D circles. A use case for 3D position estimation of spheres are for example drones that can hit back a ball they have been thrown to

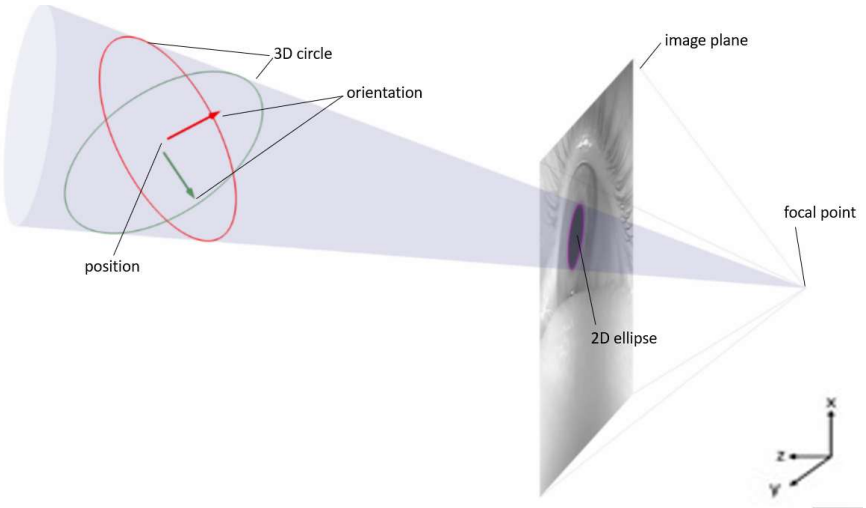



*Figure 8 Ellipse reprojection to 3D circles*

### 2.3.1 3D quadrics

The projection of a 3D circle to the image plane forms a **cone** (as could be seen in Figure 3). Safaee-Rads method works mainly with this cone. A cones implicit normal form is

$$\left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right)^2 - \left(\frac{z}{c}\right)^2 = 0 \qquad (11)$$

It is a 3D Quadric. The general form[6] of 3D quadrics has 11 terms. Its general implicit formula is

$$ax^2 + by^2 + cz^2 + fyz + gzx + hxy + ux + yv + zw = 0 \qquad (12)$$

In its matrix form, 3D quadrics have the same shape as (4)

$$(x \quad y \quad z)\begin{pmatrix} a & \frac{f}{2} & \frac{g}{2} \\ \frac{f}{2} & b & \frac{h}{2} \\ \frac{g}{2} & \frac{h}{2} & c \end{pmatrix}\begin{pmatrix} x \\ y \\ z \end{pmatrix} + (x \quad y \quad z)\begin{pmatrix} u \\ v \\ w \end{pmatrix} + d = mc^2 \qquad (13)$$

The 3x3 matrix $A$ of (13) is symmetric. The mathematics of the principal axis theorem for a cone is the same as for ellipses. Ellipses are just used as an example for PAT because it is easier to understand.

### 2.3.2 Cone transformation frames

Safaee-Rad simply uses 2D ellipses in their implicit form $ax^2 + bxy + cy^2 + dx + ey + f = 0$. From this ellipse, the cone is generated. The focal point is also the vertex of the cone (see Figure 8). Safaee-Rad provides for each coefficient of the cone a simple formula based on the coefficients of the ellipse. The overall procedure of the Principal Axis Theorem

[6] The coefficient assignment is a little different than before because Safaee-Rad uses them in the same way

**Kommentiert [MD4]:** Vllt doch nicht die coeffizienten /2 nehmen?

from 2.2 can be used to rotate the cone such that its axis are parallel to the coordinates system ones. The cones equation is much shorter now

$$\lambda_1 X^2 + \lambda_2 Y^2 + \lambda_3 Z^2 = 0 \qquad (14)$$

In order to derive an elegant analytical closed-form solution, Safaee-Rad applies several linear transformations to the cone, like PAT for example. Each transformation makes the cones implicit equation as simple as possible for the needed case. Each transformation puts the cone into a different so-called frame. In (14), the cone is in the so-called XYZ-frame. Beforehand, it was in the so-called image frame. Until now, these 3 frames are important

| | |
|---|---|
| **Image frame** | where the image plane is located on the xy-plane |
| **Camera frame** | same as image frame but positively shifted in z-direction so that the cone vertex is the coordinate systems origin. Orientation and position will be transformed to this frame in the end. |
| **XYZ frame** | where the cone is rotated so that its main axis is the z-axis |

And with it the following transformations

| | |
|---|---|
| $\mathbf{T_0}$ | Image -> Camera |
| $\mathbf{T_1}$ | Rotating to XYZ |
| $\mathbf{T_2}$ | Shifting to XYZ |

### 2.3.3 Orientation of the 3D circle

The next step is to find its 3D circles orientations (the green and red arrow of Figure 8). In order to get them, one needs to find the right conic sections that form a circle with $\text{Cone}_{XYZ}$. The mathematics of finding this very section are the most complicated part of the paper and I will try to break things down as easy as possible.

First, there is some background knowledge necessary again. One needs to know how a conic section is calculated: the equation of a given section, which is basically just a 3D plane, and the equation of the cone are set equal. The implicit equation of 3D planes is as follows[7]

$$lX + mY + nZ - p = 0 \qquad (15)$$

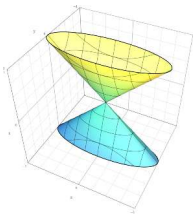


*Figure 9 Example for ConeXYZ*

with $\begin{pmatrix} l \\ m \\ n \end{pmatrix}$ as the planes normal vector named *lmn*. This vector is what one looks for at the end. It represents the orientation of the 3D circle.

The simplest 3D planes are those which are parallel to the either the *xy*, *xz* or *yz* plane of the coordinate system. For example, the plane which is parallel to *xy* plane with a distance of *p* is simply $Z = p$. With such a simple equation, the conic section can be calculated very easily. Unfortunately, it is very unlikely that the 3D circle is actually parallel to the xy plane because it probably looks similar to Figure 9.

For that reason, Safaee-Rad provides a new frame and transformation matrix $T_3$. Given *lmn*, it transforms the space so that the plane belonging to *lmn* is parallel to the xy-plane.

| | |
|---|---|
| **X'Y'Z' frame** | frame where the 3D circle is parallel to the xy-plane |
| $\mathbf{T_3}$ | XYZ -> X'Y'Z' |

[7] With regards to the XYZ frame

**Kommentiert [MD5]:** Gesucht plane lmn that forms an intersection with coneXYZ

Variable lmn wird gesucht

T3 wird angewandt ohne lmn aufzulösen

The equation oft the section ist abhängig nach lambda und lmn, aber hat form einer 2d quadrik (1)

(1) Soll einen circle ergeben dafür müssen coefficienten eine gewisse bedingung erfüllen.

Damit sie diese bedingung erfüllen, können lmn nur ganz bestimmte werte annehmen. Diese sind die gesuchhten

For now, one does not have *lmn* which means that $T_3$ consists of variables and not fixed values. The cone belonging to $T_3$ is called $Cone_{X'Y'Z'}$.

To get the conic section, $Cone_{X'Y'Z}$ is now set equal with the plane $Z = p$. Then all terms are brought to one side of the equation which results in 2D quadric.

$$aX'^2 + bX'Y' + cY'^2 + dX' + eY' + f = 0 \qquad (16)$$

All coefficients of (16) are no fixed values. They depend on lambda, *p*, and *lmn*. The exact formula of (16) can be found in Safaee-Rad's paper (p. 627, formula (20)). When (16) should represent a circle, its coefficients have to meet a few conditions. Both axis of a circle are equally long ($a = c$) and a circle has now mixed terms.

$$a = c, \qquad b = 0 \qquad (17)$$

As already mentioned, *lmn* has no fixed value yet. It can be solved so that it meets the conditions (17). Because of the two-circle ambiguity (see Figure 8), there are two possible solutions for *lmn*. Furthermore, *lmn* has a length of one, so there is the additional condition[8]

$$l^2 + m^2 + n^2 = 1 \qquad (18)$$

Kommentiert [MD6]: Soll ich hier die auflösung nach lmn noch nachrenchen?

Now, one has the orientation *lmn* set. Be aware, that vector *lmn* is still in the XYZ frame and need to transformed back to the camera frame.[9] The rotational transformation $T_1$ is used for that.

### 2.3.4 Position of the 3D circle

The position of the 3D circle is represented by its center and is the same for both solutions. The overall procedure consists of finding the circles center in the X'Y'Z' frame and then transforming it back to the camera frame. For finding the circle center in the X'Y'Z' frame $c = (X'_0|Y'_0|Z'_0)$, the normal form of a circle is needed

$$(X' - X_0)^2 + (Y - Y_0)^2 = r^2 \qquad (19)$$

*r* is the radius of the 3D circle. Pupils are light sensitive and change their size depending to the brightness. So, *r* is not known. For now, r will be initialized by an arbitrary value. The real *r* will be found in a later section. Substituting *Z'* with distance *p* (as done in 0) gives the following circle equation for the *X'Y'Z'* frame (coefficient annotation is different than before and written in capital letters)

$$AX'^2 + AY'^2 + 2pBX' + 2pCY' + p^2D = 0 \qquad (20)$$

With the method of completing the square, (20) can be rewritten to be in the same form as (19)

$$\left(X' + \frac{pB}{A}\right)^2 + \left(Y' + \frac{pC}{A}\right)^2 = \frac{p^2B^2}{A^2} + \frac{p^2C^2}{A^2} - \frac{p^2D}{A} \qquad (21)$$

Setting the right side of (21) equal with $r^2$ and solving it after *p* gives

$$p = \pm \frac{Ar}{\sqrt{B^2 + C^2 - AD}} \qquad (22)$$

[8] The norm formula usually contains a root but since the root of one equal one, there is no need for that.
[9] In the X'Y'Z' frame, vector *lmn* is simply (0 0 1)

Only the positive distance $p$ matters in this case, because only one of cone of the double cone is considered. Resubstituting $+p = Z'_0$, one has the first coordinate of the 3D circle position. The other two can be easily accessed by looking at (21)

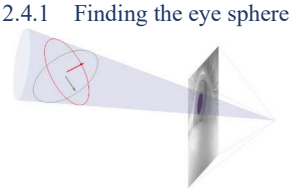
$X'_0 = -\frac{B}{A}Z'_0$ and $Y'_0 = -\frac{C}{A}Z'_0$

Keep in mind that the 3D circle position is still in the X'Y'Z' frame and has to transformed back to the camera frame.

## 2.4 3D eye model fitting

Now that the 3D circles are determined, the actual eye model fitting can begin. The second paper "A fully-automatic, temporal approach to single camera, glint-free 3D eye model fitting" by Swirski et al. (Swirski et al., 2013) uses the 3D circles to approximate the 3D eye model. The 3D eye model basically hast just two parameters. The eye sphere center in relation to the camera and the radius of the eye sphere.

### 2.4.1 Finding the eye sphere center



The eye ball center is the most important part of the eye model. For finding the eye sphere center, the two-circle ambiguity has to be dissolved first. Swirski uses a simple but clever trick for that. The 3D circles gaze vectors are projected to image plane again. The gaze vector is the orientation vector with the position as the position vector. The gaze vectors of both the red and the green circle form the same vector on the image plane. The lines for all pupils recorded during a session will be intersected to find the 2D eyeball center. Since it is very unlikely that all vectors will actually intersect in one point, the point with the least-square distance to all vectors will be the actual 2D eye center. This eye center is than reprojected in 3D (see Figure 10).

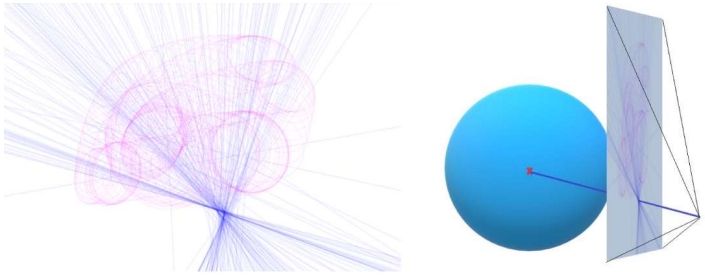


*Figure 10 The eye ball center is determined by the intersection of the gaze vectors and then reprojected to 3D*

### 2.4.2 Resolve two circle ambiguity

Now that the eye center $c$ is available, the two-circle ambiguity can be really dissolved. One of gaze vectors points towards the center whereas the other one points away from it. The one that points away from the sphere center is the actual one (see Figure 11).

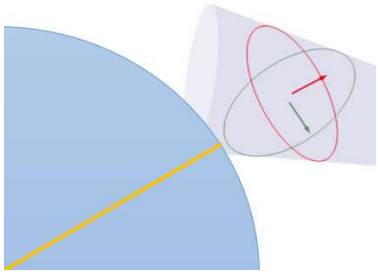


*Figure 11 The red circle is the actual one in this case*

Mathematically, this is calculated with the dot product with $c$-$p_i$ as the yellow line in Figure 11. If the scalar product is greater than zero than the pupil normal points away from it.

$$n_i * (c - p_i) > 0 \qquad (23)$$

## 2.5 Resolving the distance-size ambiguity

**Kommentiert [MD7]:** Erkläre genau wieso es die distance size ambiguity gibt
Und den willkürlichen 3d radius

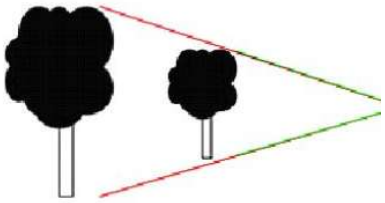


*Figure 12 Example for the distance-size ambiguity*

It is unknown by the 2D ellipse alone, whether the pupil is close to the camera and small or rather far away and big. That is why the 3D circles radius in section 2.3.4 “Position of the 3D circle” was initialized arbitrarily. This is the so-called **distance-size ambiguity**.

Now that the eye center is found and the two-circle ambiguity is dissolved, the conditions are met to resolve this ambiguity. The actual 3D pupil position is the intersection between the yellow and blue line (see Figure 14 Finding the actual pupil position Figure 14). As the figure already shows, the blue lines position vector is the 3D eye sphere center. Its orientation vector is the pupils orientation. The yellow line is the projection line of the pupil position. It is therefore defined by two points: the focal point and the current 3D pupil position. Both lines will probably not really intersect. The least square distance intersection is used again (like done in 2.4.1 Finding the eye sphere center).

Now both ambiguities are resolved and the actual 3D pupil can be outputted to the user.

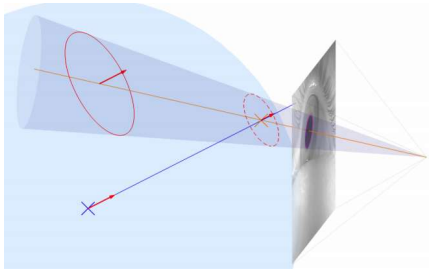


*Figure 14 Finding the actual pupil position*

# 3 Implementation process

Most of the time was spent with coding the above described concepts in Python. This code could will be from now on referred to as the implementation. The implementation can be found in the appendix or with GitHub (github.com/drduda/ba_thesis). The whole implementation can be described by a pipeline. First, there is the projection simulation (Section 2.1) then there is the elliptic reprojection (Section 2.2) and then the actual eye tracking (Section 2.4). All calculations in this pipeline are built upon one another. Just one bug or mistake is enough so that the code's calculated results are completely wrong. The following section is the story of the implementation process. This story is mostly about two topics: software engineering and geometrical mathematics.

The code is about 600 lines long. Without a clear code structure, it is easy to lose the overview. And without overview, it is rather easy to disimprove things. It will be described how the overview was kept with the methods of modern software engineering.

There were a few mathematical obstacles too. It was not work with a to the paper corresponding code base. The papers from Safaee-Rad and Swirski alone are not always detailed enough to actually rebuild their methods. To overcome these issues, own assumptions have been made and tested.

## 3.1 Implementing the projection

The first step is to simulate the two-dimensional elliptic pupil contours which serve as the input data for the actual eye tracking.

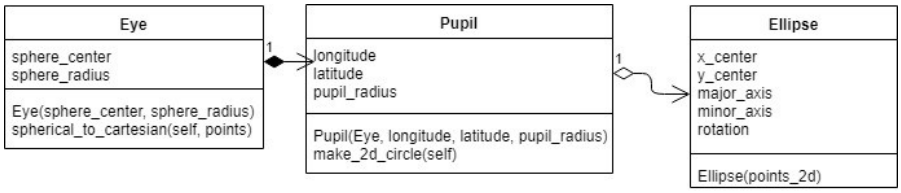



*Figure 15 UML of generating the input data*

Above one can see the UML class diagram for generating these ellipses. Classes are written with capital letter in this thesis. First, *Eye* is initialized as a singleton object. Its radius is set to 1.2cm which is the average size of an eye. The *Eye's* center describes the in reality unknown positional relation to the camera. It is therefore set arbitrarily. Then *Pupil* is initialized. Its parameters describe a 3D circle sitting on the surface of the eye sphere.

Now, one has to calculate the points that lay on the pupils' 3D circle. These points will be projected to the image plane and serve as the input for the *Ellipse* constructor (see Figure 16).

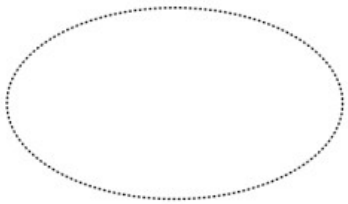


*Figure 16 Example for points_2d , the input of the ellipse constructor*

**Kommentiert [MD8]:** Parameters of pupil describe a 3d circle
Calculating 3d points that lay on this 3d circle
How?
Make 2d circle with formula
Imagine this 2d circle lays on the map of the earth
Map of the earth is 2d representation of a 3d object
Coordinates on world map are long and lat
They are spherical coordinates
Spherical to cartesian
Trigonometrie

Now 3d circle is there
Get projected to image plane

How are the points that lay on the 3D circle are generated? First, a simple circle in two-dimensional space is generated. There are two different forms for representing geometrical objects. Until now, only the implicit form was introduced.[10] For generating points laying on the surface of geometrical objects, however, the parametric form is more suitable. Parametric formulas have an additional parameter $\theta$. For a circle, it is as follows for the x- and y-coordinate

$$x = r * cos(\theta) + x_{Center} \; with \; \theta \in \{0, 2\pi\} \qquad (24)$$

$$y = r * \sin(\theta) + y_{Center}$$

Now, this two-dimensional circle has to be projected onto the three-dimensional surface of the eye. In order to do that, one has to imagine that this circle lays on the map of the earth. The map of the earth is a 2D representation of a 3D object. Its coordinates are described by the spherical coordinates' longitude and latitude. In equation (25) *x* stays for the longitude and *y* for the latitude. $(x|y)_{Center}$ is the long- and latitude and *r* is the radius of *Pupil*. With the method Eye.spherical_to_cartesian, theses spherical coordinates will be now be transformed to 3D cartesian coordinates laying on the surface of Eye. Then, they will be transformed with persepective projection to the image plane and the result could be look like Figure 16.

The constructor of *Ellipse* uses the method fitEllipse which returns corresponding parameters of *Ellipse* like major axis and minor axis length, rotation and the ellipse center. fitEllipse is imported from the OpenCV. A program library for computer vision.

Programming the projection module have been my first steps in object-oriented programming. The code does not look as clean as the modules I later programmed. This made the debugging of this module a truly nerve-wracking and tedious experience. Even spotting simple bugs like a wrong measurement unit, took a lot of time.

**Kommentiert [MD9]:** Darf man das so schreiben?

## 3.2 Implementing the ellipse reprojection

It was clear that I needed to change my way of working and coding. Otherwise, I would not be able to make ellipse reprojection work. In the result section, Safaee-Rad, author of the ellipse reprojection paper, provides detailed calculation steps of his method with a concrete example. This example calculation, from now on referred to as **experimental results**, was very helpful for understanding his method. Furthermore, these values could be used for writing unit tests. From now on, unit tests were written before the actual code. This way of coding is called Test-Driven Development (TDD). TDD often helped to spot bugs right after they were made.

### 3.2.1 Simplifying the cone transformation

Before coming to the implementation of the ellipse reprojection, there is something interesting about the cone transformation process in general. This process could perhaps be simplified. The process for getting the cone in X'Y'Z' frame is described as follows by the paper

$$Ellipse_{Implicit} \rightarrow Cone_{Image} \xrightarrow{T_1+T_2} Cone_{XYZ} \xrightarrow{T_3} Cone_{X'Y'Z'} \qquad (25)$$

$Cone_{Camera}$ is just $Cone_{Image}$ with the vertex shifted to the coordinates system origin. Replacing $Cone_{Image}$ with $Cone_{Camera}$ could make $T_2$ (the shifting transformation) obsolete. But first one needs to ask oneself on how to get the cone in the camera frame. This should be possible, simply by leaving out the non-quadric terms $ux + yv + zw$ of $Cone_{Image}$. By looking at the experimental results of Safaee-Rad (p.632) where the implicit forms of cone in all its transformation steps are shown, one sees that that this is indeed correct.

[10] $x^2 + y^2 - 1 = 0$ is the implicit form of the unit circle

The position of the 3D circle (see 2.3.4) needs to be transformed from *X'Y'Z'* back to the camera frame. This is made possible by the total transformation

$$T_{Total} = T_0 T_1 T_2 T_3 \qquad (26)$$

$T_2$ is calculated by the shifting of $Cone_{Image}$. Without it, the total transformation is not complete. However, $T_0$ is also just a shifting not a rotational transformation. Maybe by leaving out $T_0$ and $T_2$, the total transformation can be calculated, too. And indeed, testing the assumption with the experimental results shows that this hypothesis is indeed correct.

Kommentiert [MD10]: Verweise auf den GitHub Code

$$T_{Total} = T_0 T_1 T_2 T_3 = T_1 T_3 \qquad (27)$$

The proof for (27) can be found in the implementation (module tests.test_simplify_transformation, line 27). $T_0$ and $T_2$ just shift the cone back and forth. The cone transformation can be radically simplified.

### 3.2.2 Designing it in an object-orientated way

The projection module was programmed without making an UML diagram before. This turned out to be a mistake because this made the code messy and it took a lot of time afterwards to actually make the module work. So, for the ellipse reprojection a UML diagram was made.

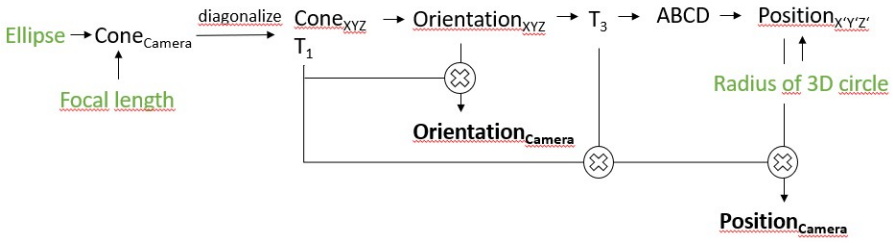



*Figure 17 Calculation sequence of the ellipse unprojection*

The first step is to visualize the calculation sequences of the ellipse reprojection (see Figure 17) in order to know in detail what exactly happens. The circle with the x inside represents multiplication. The inputs are green and the outputs are marked bold. The radius of the 3D circle is not known yet and will be an arbitrary value. The two-circle ambiguity is not solved, so there are actually two orientations as an output.

The principles of object orientated programming served as orientation for designing clear, readable and maintainable code. The *Principle of sole responsibility* says that every entity[11] is responsible for one thing or task. In practice that means that almost all of the calculation steps in Figure 17 have their own class. This helps to improve the overview since all of the detailed calculations are encapsulated inside this class (different layer of abstraction).

[11] An entity can be a module, class, instance etc.

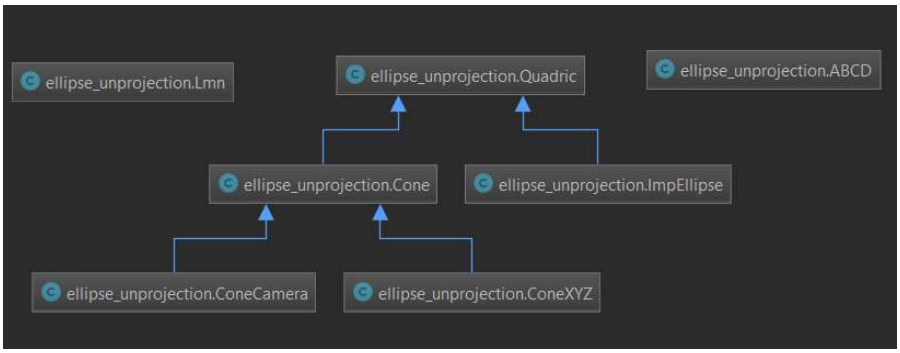



*Figure 18 Ellipse reprojection UML class diagram*

The different classes are quite similar, so they inherited a lot from each other (see Figure 18, a more detailed class diagram is in the appendix). All quadrics are in their implicit form. Unfortunately, no Python library has a tool for visualizing implicit forms in 3D but visualizing intermediate steps would be really helpful to understand what the algorithm is actually doing. Wolfram Alpha, an online mathematics engine, supports 3D visualization of implicit equations. The method Quadric.get_equation() was implemented to print out the corresponding implicit equation that belongs to a Quadric object. Because of inheritance, all of its subclasses have this method, too. Quadric.get_equation() works like this: below one can see *Cone* attribute for the coefficient *a* of the term $ax^2$ with the value of 5.

```
a_xx = 5     =>     "5*xx"
```

By having the variable $x^2$ or rather xx inside the attribute name a_xx, Quadric.get_equation only needs string manipulation in order to be working. The string "5*xx" will be used as part of the output of Quadric.get_equation().

## 3.3 From the parametric to the implicit ellipse form

Safaee-Rad only works with ellipses in their implicit form. Computer vision libraries like OpenCV usually return ellipses by their parameters. Therefore, a method for converting an ellipse by its parametric to its implicit form is needed.

**Kommentiert [MD11]:** https://math.stackexchange.com/questions/426150/what-is-the-general-equation-of-the-ellipse-that-is-not-in-the-origin-and-rotate

First, a clear interface for the ellipse parameter needs to be defined. As an orientation served the OpenCV ellipse interface. It has five components: the major and minor axis length, x and y center and the rotation of the major axis in degrees. The major axis lays on the x-axis, the ellipse is then rotated mathematically positive (clockwise). By its nature, the major axis is bigger than the minor one and the rotation only makes sense up to 180°. Sanity checks proof if these conditions are fulfilled. The following formula

$$\frac{\big((x-x_0)\cos(rot)+(y-y_0)\sin(rot)\big)^2}{major^2}+\frac{\big((x-x_0)\sin(rot)-(y-y_0)\cos(rot)\big)^2}{minor^2}-1=0 \qquad (28)$$

provides the implicit function but with ellipse parameters as variables. The simplified, factorized form of (28) looks like the usual implicit quadric function (1). The coefficients of the factorized function are what is looked for. In order to factorize (28) and get the coefficients, SymPy, a Python library for symbolic calculation, is used.

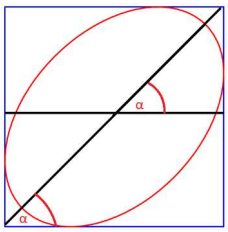



*Figure 19 The line between the vertices can be uses to derive the rotation*

The paper by Safaee-Rad et. al only works with implicit ellipses and does therefore not provide any example calculation for converting ellipses from parametric to implicit. Fortunately, Wolfram Alpha can calculate the parameters of an ellipse based by its implicit form. These parameters can be used for writing unit tests. However, Wolfram Alpha does not show the rotational parameter of an ellipse. It only shows the vertices of the rectangle around the ellipse. The line between these vertices can be used to derive the rotation of the ellipse (see Figure 19).

Five parameters are needed for a unique representation of an ellipse. However (1) has six coefficients, one of them only scales the other coefficients and is actually superfluously. This makes unit tests rather difficult because there is no unique representation to compare to. Therefore, all coefficients are divided by $e$ in order to get a unique representation (29).

$$ax^2 + bxy + cy^2 + dx + ey + f = a'^{x^2} + b'xy + c'^{y^2} + d'x + y + f' = 0 \ \ with\ a' = \frac{a}{e} \qquad (29)$$

## 3.4 Defining a clear interface for the eye tracking input data

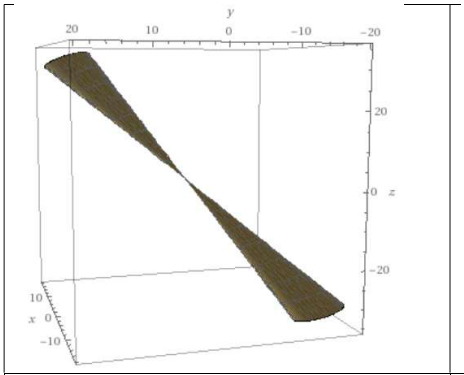
In the last section, the interface for the ellipse was defined. Now, the data structure for the input data which consist of all the tracked ellipses needs to be defined. Each ellipse is represented by a dictionary like

```
{"x_center": 0.2, "y_center": 0.3, …}
```

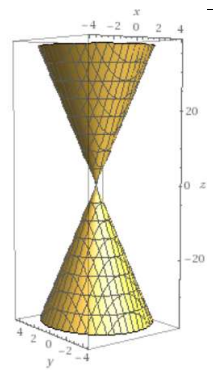
And all these ellipses are saved in a list. The only use case of this implementation so far, is with simulated input data. In order to be useable with real world input data, it would be desirable that the input data structure is independent from Python and is interpretable by humans. JSON is the ideal file format for these requirements. All major programming languages can create and load a JSON file from lists and dictionaries. JSONs are saved as a string and are therefore readable for humans, too. Therefore, it was implemented that of the eye tracking algorithm can receive JSON objects as input data, too.

## 3.5 Diagonalizing $Cone_{Camera}$ to $Cone_{XYZ}$

Section 2.2.1 “Rotation” describes how the rotation of an ellipse works so that its major and minor axis are parallel to the coordinate system’s axis and no mixed terms occur anymore. In this section, the rotation (and its implementation) will be applied for a cone as described in section 2.3.1 “3D Quadrics”.




*$Cone_{Camera}$*




*$Cone_{XYZ}$*

As described in the section “Simplifying the cone transformation pipeline”, $Cone_{Camera}$ instead of $Cone_{Image}$ will be used for rotation. It has the following form.

$$ax^2 + by^2 + cz^2 + fyz + gzx + hxy = 0 \qquad (30)$$

To diagonalize (30), its coefficients will be put in a symmetric matrix $A$. To actually diagonalize $A$, the method numpy.linalg.eig for calculating the eigenvalues and eigenvectors was used. This method, however, has a crucial shortcoming. When returned, the eigenvalues and eigenvectors are not necessarily ordered. The eigenvectors are the column vectors of the transformation matrix $T_1$. Changing their order changes the algebraic sign of the determinant of $T_1$. That is why putting them in the right order is not as simple as it sounds. Fortunately, NumPy has another method for diagonalizing. The method numpy.linalg.eigh is only for diagonalizing symmetric matrices and return the eigenvalues and eigenvectors in an ordered way.

The eigenvalues make the coefficients λ of $Cone_{XYZ}$

$$\lambda_1 X^2 + \lambda_2 Y^2 + \lambda_3 Z^2 = 0 \qquad (31)$$

Compared with the results of Safaee-Rad, the $T_1$ looks a little different then his one. The difference can be explained with the following matrices.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \qquad (32)$$

Of course, T1 looks different. These are just simple examples to make the case. The left matrices of (32) is my first solution whereas the right is the correct one. As already mentioned, the columns are the eigenvectors of $A$. Since the eigenspace is the span of the eigenvector, both matrices are valid in terms of this condition. Safaee-Rad says that the column vectors must satisfy the right-hand rule, but both of them do. They even have the same determinant which is 1. It can be said that they are both “correct” solutions for $T_1$. Actually, there are four different “correct” matrices.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \quad \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \qquad (33)$$

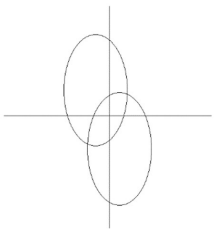


*Figure 20 Both quadrics are correctly rotated*

All with the same determinant, all fulfill the right-hand-rule, only the column vectors have a different algebraic sign. The ellipse example of Figure 20 could illustrate what the difference between the matrices of (33) is. [12] Both ellipses are rotated such that their axis are parallel to coordinate systems axis. The principal axis theorem is correctly applied. The same can probably said about (33). Probably, they all rotate the cone to its standard form but they do it in a different manner.

So why not arbitrarily chose one of the four matrices? Given one point, each of these transformation matrices will transform this point to another location. For each tracked pupil, there are four different $T_1$. Choosing one matrix arbitrarily each time would be like choosing different units of measurement each time. The problem here is that between different tracked pupils, one does not know which “measurement unit” is which.

[12] This is just an assumption.

Safaee-Rad provides another way for calculating $T_1$, too. It is based on the discriminating cubic, an algebraic function for finding zero values of a polynomial. The discriminating cubic function only allow positive values for the second row of T1 (Safaee-Rad et al., p.626f). This would lead to a determinant of negative one which would be wrong, too. In the eyperi results, Safaee-Rad simply changes the algebraic sign of the last column vector of $T_1$ to make the determinant 1 again.

So, in the implementation, the algebraic signs of the column vectors of T1 will be changed such that the values of the second row will be positive. If the determinant is then negative one, the algebraic sign of the last column vector will be changed again to make the determinant positive again. Later tests will show whether this approach is correct.

## 3.6 Project to 2D to find intersection

*Figure 21 Left example is correct*

The next step is to project the gaze vectors to the 2D image plane again. So that the 2D eyeball center can be found. There are two conditions that have to be fulfilled during the implementation. Frist, the projected gaze vectors of both the red and the green 3D circle of one pupil have to form the same line (see Figure 21). When this is fulfilled the second condition has to be meet: the 2D gaze line of all pupils tracked during a session have to intersect (see Figure 10 as a reminder). This intersection makes the 2D eyeball center. Meeting this condition has the character of integration testing. Integration tests are one level above unit tests. They test whether different units work correctly together. In this case, this means that the whole pipeline from the simulation other the ellipse reprojection to finding the eyeball center would work correctly together.

The gaze vector $\vec{g}$ has the 3D circle's orientation as its orientational vector $\vec{o}$ and the 3D circles position as its position vector $\vec{p}$. It was like this projected to 2D

$$\vec{g}_{2D} = f_{2D}(\vec{p}) + r * f_{2D}(\vec{o}) \quad with\, r \in \boldsymbol{R} \qquad (34)$$

$f_{2D}$ is the function for projecting 3D points to the image plane. Unfortunately, the red and green gaze vector did not form the same line. It was thought that this is due to the problems that there have been with calculating T1.

### 3.6.1 The wrong conclusion

This subsubsection is about a wrong conclusion made during the debugging process of Section 3.6 "Project to 2d". The gaze line of the red and the green circle should form the same line when projected to the image plane but they did not. It was thought that this is due to the problems that have been described in Section 3.5. "Diagonalize $Cone_{Camera}$".

For the green circle, the result by Safaee-Rad were used. He did not provide the red circles. So, the red circle's gaze line was calculated by this implementation which means that it could be wrong due to the problem of finding the right order and column signs for T1 (see Section #todo).

As mentioned in #todo, there are 4 different configurations of T1 due to the column sign possible. The idea now is to try each of these 4 transformations: once for the red and green gaze lines orientation in order to look, which of these different configurations lead to the same projected gaze line for the red as well as the green circle. The table below shows the result.

| Configuration | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | No | Same line | Same line | No |
| 1 | Same line | No | No | Same line |
| 2 | Same line | No | No | Same line |
| 3 | No | Same line | Same line | No |

Surprisingly, if the red and green circle use the same transformation T1, they will never work together. As a consequence, the 3D circle position has to use the two different T1 which results in two different 3D circle positions for the positive and negative case.

Taking a closer look at the results of the table reveals that each time, red and green form the same line on the image plane, it is only because they already formed the same line in 3D space. This cannot actually be which made it obvious that the error has to be somewhere else.

### 3.6.2 The right conclusion

#todo lösch 4.3

**Fehler! Verweisquelle konnte nicht gefunden werden.Fehler! Verweisquelle konnte nicht gefunden werden.**The error was somewhere else. The actual 2D gaze is found by the following formula.

$$\vec{g}_{2D} = f_{2D}(\vec{p}) + r * f_{2D}(\vec{o} - \vec{p}) \quad with\, r \in \boldsymbol{R} \qquad (35)$$

In (35), two points from the 3D gaze line were taken ($\vec{p}$ and $\vec{o} - \vec{p}$).[13] Those two points are then projected to the image plane. Both the red and the green 2D gaze vector form the same line. Doing a unit test for the projection of the gaze line to the image plane would have prevent the wrong conclusion of #todo. Taking the input data from Figure 6 and projecting its gaze vectors to the image plane resulted in Figure 22.

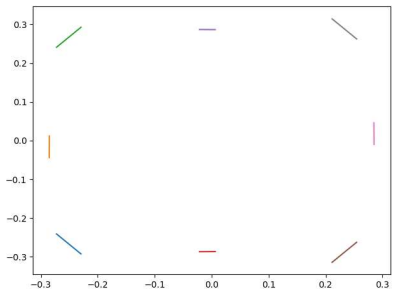



*Figure 22 First, wrong example of gaze vectors*

Clearly, all those lines do not intersect in one area. They all have to be rotated by 90 degrees. Why is this the case? In the projection module, the OpenCV library version 3 was used. However, the OpenCV documentation for version 2 was unintentionally used to define the ellipse interface. Between this version they changed the ellipse API.

Instead of downgrading to the OpenCV version of the documentation. A simpler approach was used. The rotational parameter was rotated by 90 degrees with the following formula.

$$rotation = (rotation + 90)\ mod\ 180 \qquad (36)$$

Now, the gaze lines of all pupils are intersecting in one area. The result will be discussed in the result section in further detail. The integration test is passed. Everything that was implemented so far works correctly together.

The gaze lines will almost certainly not intersect in one point. Instead the point closest to each gaze line in least-square sense will be calculated by the formula below. $\vec{o}$ and $\vec{p}$ are in their two-dimensional form.

$$center_{2D} = \left(\sum^{i} I - \vec{o}_i\, \vec{o}_i^{\,T}\right)^{-1} \left(\sum^{i} (I - \vec{o}_i\, \vec{o}_i^{\,T})\vec{p}_i\right) \qquad (37)$$

A unit test with a simple example of just two intersecting lines was created to test the implementation of (37). This unit test revealed that in order to let (37) work correctly, the two-dimensional $\vec{o}$ has again be normalized to a length of one.

[13] Vectors also represent points.

For testing the tracking performance under noisy input data (as will be done in the result section), the tracked eye ball center will be compared to its ground truth. That is why the rest of Section 2 "Concepts" like resolving the distance-size ambiguity do not have to be implemented.

The result of this implementation will be discussed in the "Results" section.

### 3.7 Architecture

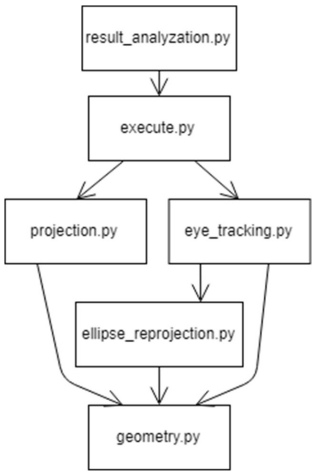



*Figure 23 Architecture of implementation*

This section gives a brief description about the module structure/ architecture of the implementation. Each of the rectangles in Figure 23 describe a python file/module. The arrow means that ellipse_reprojection.py imports geometry.py. The modules in Figure 23 are structured in layers. Each layer comes with a different level of abstraction. geometry.py for example implements a lot of the already discussed geometrical methods and classes like *ParametricEllipse*. It is in the lowest layer of abstraction. Execution.py connects the simulation of elliptic input data with the actual eye tracking algorithm. Parameters like eye_center etc. are set in this module. This module also calculates the distance between the actual and the tracked eye center. result_analyzation.py has not introduced yet. It is used for Section 5 "Results". This module adds noise to the input data in order to examine how robust the eye tracking algorithm is.

## 4 Alternative approaches

Research and development are no a linear process. This thesis is no exception from that. This section is about the alternative approaches and wrong tracks that have been tried out on the way to a working implementation – for example alternative algorithms for the elliptic reprojection and the 3D eye model fitting. At the beginning of this thesis, the mathematics of the actual concepts of section 2 were considered as too complicated.

### 4.1 Approximation of the elliptic reprojection

This section is about finding an alternative method for the elliptic reprojection suggested by Safaee-Rad. There are two steps for finding the 3D-circle of the elliptic reprojection. First, a linear equation system is used to calculate the coefficients of the cone (the one of Figure 8). Then, different intersection planes will be tried in a brute-force manner to find the section that forms a circle with the cone.

Kommentiert [M[12]]: rot ist ellipsen center
blau ist pupillen center

#### 4.1.1 Finding the corresponding cone

A cone is a three-dimensional quadric, its implicit form has nine coefficients. Therefore, a linear equation system with 9 equations are needed at least in order to get a unique solution set. Each equation has the following form.

$$ax^2 + by^2 + cz^2 + fyz + gzx + hxy + ux + yv + zw = 0 \quad (38)$$

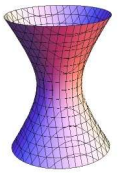


*Figure 24 Example for one-sheeted hyperboloid*

For x-, y- and z-value , every point of the cones surface can be considered. The simplest approach is to put 8 points of the ellipse plus the vertex point with point (0|0|0) in a homogenous linear equation system. The resulting solution set still contains two variables (or rather two degrees of freedom). Since a cone is point-symmetric, three of the points can be multiplied with minus 1 in order to get their mirrored counterpart. Putting them in the linear equation system results in the geometric figure that can be seen in Figure 25. The figure is visualized with Wolfram Alpha. Wolfram Alpha displays that the geometric figure is a one-sheeted hyperboloid and not a cone. The intersection of the xz-plane reveals that the figure has indeed no real vertex. Wolfram Alpha is right. It is actually a one-sheeted hyperboloid and not a cone.

**Kommentiert [MD13]:** -7286A922051387.84*x**2 - 1.0*x*y + 0.500000000000001*x*z + 0.103553390593276*x - 437215323083265.0*y**2 + 466363011288819.0*y*z - 2.0*y - 116590752822205.0*z**2 + z

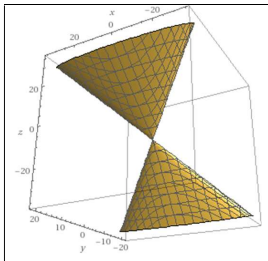
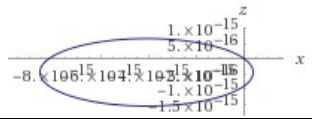


*Figure 25 Result of the linear equation system*



*The intersection of the xz-plane.*

Next, a simpler equation that still can represent a cone is used for the linear equation system.[14]

$$ax^2 + by^2 + cz^2 + fyz + gzx + hxy = 0 \qquad (39)$$

The solution set is still a one-sheeted hyperboloid but it might in practice be functioning like a cone. Next, the right plane that forms a circle when intersected with the "cone" needs to be found. Figure 26 is a cross section that illustrates an approximation method for finding the right intersection works. Between each plane (the red ones) is roughly a rotation of 15 degrees. The intersection between the "cone" and plane forms an ellipse. The intersection whose ellipse is most similar to a circle is selected. Arround this selected intersection are then again planes, the green ones, initialized. This process will be repeated until the ellipses are almost like a circle.

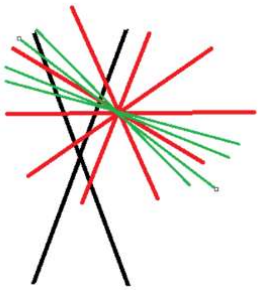


*Figure 26 Illustration of the approximation algorithm*

This rotational procedure has to be made arround the x as well as the y axis. The implementation of this method did not work for unknown reasons. Propably because of the pairing of two rotations at the same time or because some of the section did not form ellipses but parabolas. It turned out that this method is not as simple as it tought it was.

[14] The implicit form of $\text{Cone}_{\text{Camera}}$

## 4.2 Shadow mapping with genetic algorithms

The next approach uses shadow mapping for finding the eye ball center. This approach can be illustrated with the following construction: a light source shines from the focal point onto the image plane. The edge of the 2D ellipses are cut out of the image plane so that light can shine trough. Behind the image plane is a sphere. This sphere has the size of an eye ball and the shadows of the ellipses shine onto this sphere. When the sphere is at the right position, the shadows of the ellipses form circles.

### 4.2.1 Finding the right sphere position

Genetic algorithms were chosen as the favored optimization method for finding the best eyeball position. They consist of following sequences

- Initialize the first generation of solution candidates
- Each solution candidate will be values by an evaluation function
- Following steps are repeated until a termination condition is fulfilled
    - Selection of the solution candidates for reproduction (the parents)
    - Recombination of the parents (mating)
    - Random changes of children (mutations)
    - Evaluation of children's
    - Selection of the new generation (survival)

When designing genetic algorithms, the most critical part is to find a good evaluation function. In this case, it would evaluate how similar the "ellipses' shadows" are to a circle. However, there are few edge cases to be considered for the evaluation. How should it be evaluated when the "ellipse shadow" does not hit the eyeball or hit it just partly? Does the eyeball tend to come as close as possible towards the image plane because error tolerance there is the biggest? If so, should it (or rather how much should it) be punished when the eyeball gets closer to the image plane? Because all of this open questions, this approach was eventually discarded. Its source code can be found in the appendix under genetic_algorithms.py.

# 5 Results

This section is about the results gained by the implementation of the eye tracking algorithm. These results will be compared to the ground truth data so that the eye tracking accuracy can be analyzed. Then noise will be added to the 2D ellipses that serve as input data. This is done to examine how robust the eye tracking actually is. Adding noise changes some aspects of the implementation. Some sanity checks had to be removed for example. That is why there is a new Git branch called result_analyzation extra for this section. The experiments and figures expressed in this section can be repeated with the result_analyzation.py module.

## 5.1 Performance analyzation

At first, the result looked like the Figure 27 below. X- and y- axis are scaled in centimeters. The optical axis of the eye tracking camera (which basically means directly in front of the camera) is the origin (0|0) of coordinate system.[15] The grey points represent the pupil's position and the grey lines are corresponding gaze vectors. The gaze vectors have different lengths because of perspective distortion. The short vectors are short because they show towards the camera and the long ones are long because they show away. The 'o'-marker is the actual projected eye center (the ground truth) and the 'x'-marker is the one calculated by the eye tracking algorithm. The performance (or the measurement error) is the difference between both centers. It is 0.14 cm in this case.

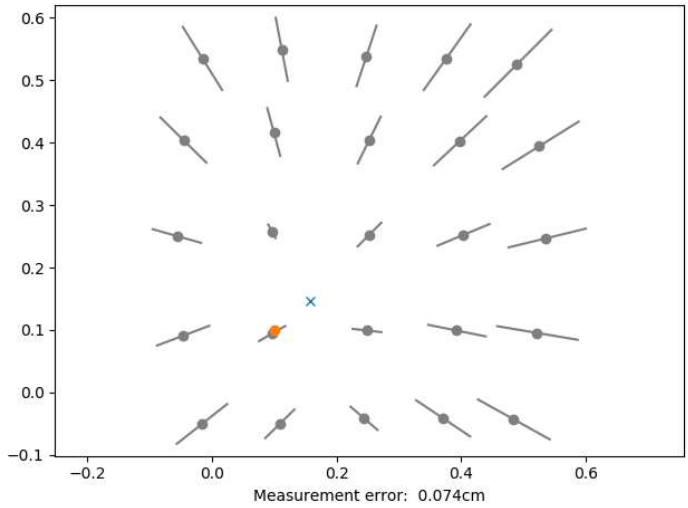



*Figure 27 First result of the implementation*

There was an implementational error in the result of figure #todo. The "projection to 2D" functions had different focal lengths as a parameter. Adjusting them all to the same value improved the result significantly as could be seen in the figures below. The measurement error is roughly a tenth of before.

[15] Which is just the "project to 2D" function in this case

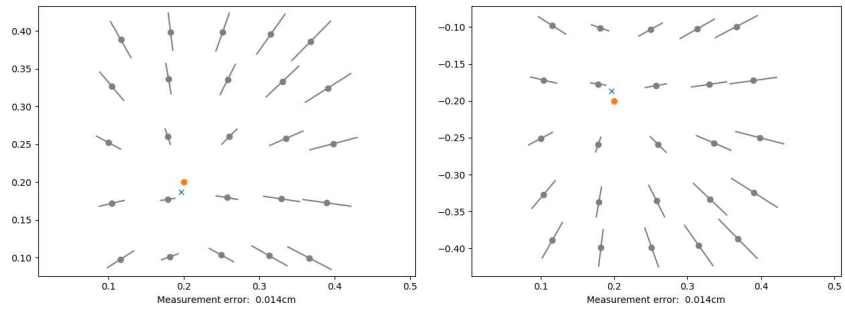



*Figure 28 The corrected implementational results*

Taking a closer look at the gaze vectors on the left plot of Figure 28, one notices that all vectors above the projected eye center are exactly showing towards the actual center ‘o’ The ones below are slightly mistaken, not all of them show exactly to the center. On the right plot, the opposite is the case. The eyeball is below the x-axis. All y-values are negative. The gaze vectors above the eye center are not as accurate as the one below. What happens when all inaccurate gaze vectors are removed? The measurement error is just the half with 0.08 mm as could be seen in Figure 29. Nonetheless, the result is still not perfect.

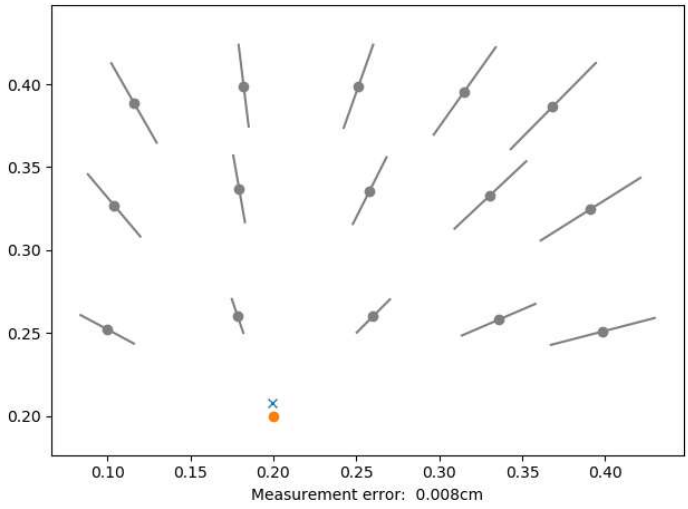



*Figure 29 Just the accurate gaze vectors are taken*

### 5.1.1 Position based analyzation

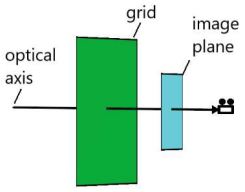



*Figure 30*

The measurement error of the eye tracking is slightly dependent on the eye ball's position. When the eyeball position is exactly in front of the camera for example, there is no measurement error at all (see Figure 33). In order to structurally analyze how the eyeball's position effects the performance, the eye ball was moved above a grid. This grid is parallel to the image plane. For each position on the grid, an eye tracking session was simulated and the performance (the measurement error) was recorded. Figure 30 illustrate the experimental design. Figure 31 illustrates the result of this positional performance analysis. The x and y axis represent the position of the eye center on the grid whereas the redness represents the measurement error.

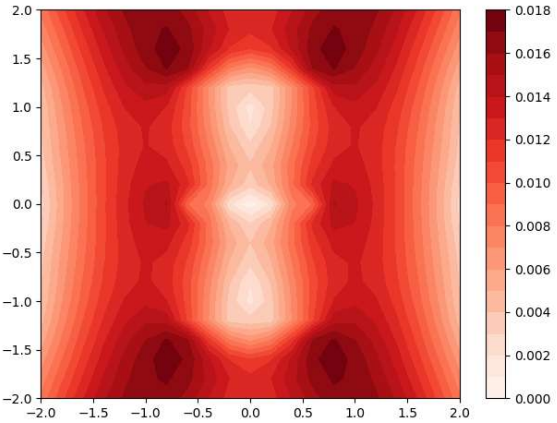



*Figure 31 Performance analysis depended on eyeball position*

The resolution is 40 x 40 which makes 1600 simulated eye tracking sessions. As expected, the optical axis of the camera (the coordinates system origin) has the best performance. The maximum is with 0.18 mm still in a tolerable level. It was expected that the only relevant factor for the eyeball positional based performance would be how far the eyeball is to the camera's optical axis. It should not matter performance wise whether the eyeball is a centimeter above or left the optical axis. This would result in circle shaped contours for Figure 31. Obviously, this is not the case here. At least Figure 31 is symmetric about the x- and y-axis. This means semantically, that it does not matter performance wise whether the eyeball is one centimeter above or under the optical axis. However, it remains an open question why the eye tracking's positional performance is like it is.

## 5.2 Head slippage

eyeTrax uses eye tracking in VR. The eye tracking camera is built in the VR headset. VR headsets are not light so when the VR headset's headband is too loose, the headset and with it the camera can move on the user's head. This phenomenon, called head slippage, is a common problem for eyeTrax. In this section, head slippage is simulated. In order to do so, two input data sets with different actual eyeball positions are merged and hand over to the actual eye tracking algorithm. Figure 32 illustrates the results. As expected, the tracked eye center is between the two actual ones.

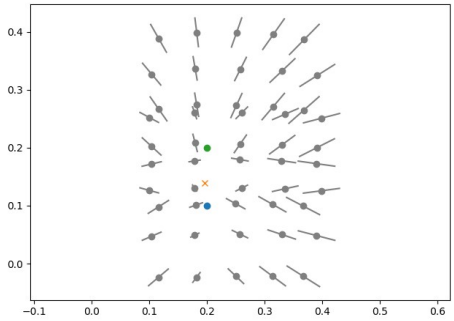



*Figure 32 Head slippage simulation*

**Kommentiert [MD14]:** sess1 = exe.Session(eye_center=[1, -5, 1])
sess2 = exe.Session(eye_center=[1, -5, 0.5])

## 5.3 Performance with different sized pupils

Pupils changes their size depended to the brightness. All eye tracking algorithms have to consider different sized pupils. So far only same sized pupils have been simulated. In this subsection, different sized pupils will be used.

The distance-size ambiguity does not to be resolved for finding the eye sphere center which means that there is no obstacle to work with different sized pupils. And indeed Figure 33 shows that there is no performance issue when working with different sized pupils. Additionally to perspective distortion, the gaze vectors now have different lengths due to different sized pupils.[16]

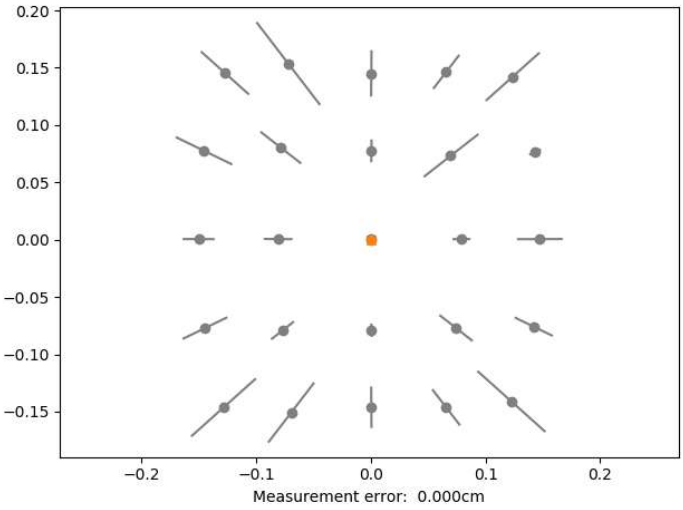



*Figure 33 Different scaled pupils*

[16] At this stage, the eye tracking algorithm considers big pupils as being further away which makes there gaze vectors smaller. The distance-size ambiguity will be resolved in a later stage (see Section 2.5)

## 5.4 Noise analyzation

In the next experiments, there will be some noise added to the 2D ellipses. Since the ground truth data is still available. It can be analyzed how well the tracking behaves with regards to noisy data.

The noise will be generated over a normal distribution with a mean of 1. The standard deviation is an independent variable of the experiment. 5% standard deviation will be annotated as 5% noise. The noise will then be factorized with the parameters of the ellipse. Each parameter gets its own noise taken from the normal distribution.

As can be seen in Figure 34, more noise lead to more "chaotic" data but does not necessarily lead to a worse performance.

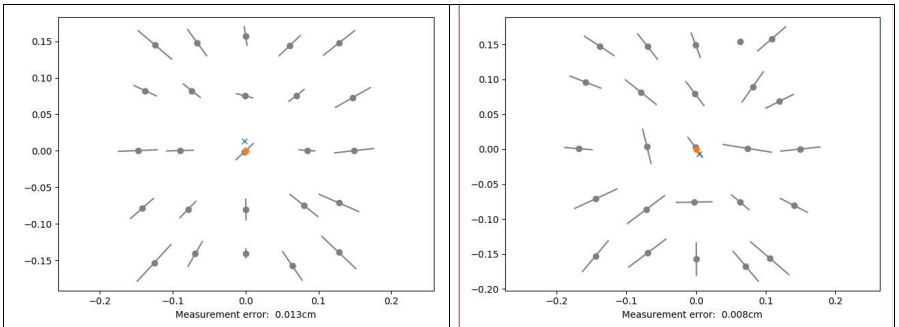



*Figure 34 The left plot has 5% the right one 10% noise*

Kommentiert [MD15]: 10% standartabweichung

How does the eyeball's position and noisy data interfere with each other? Will the same pattern as from Figure 31 occur again or will there be a new pattern? The positional based experiment from before will be repeated with noisy data.

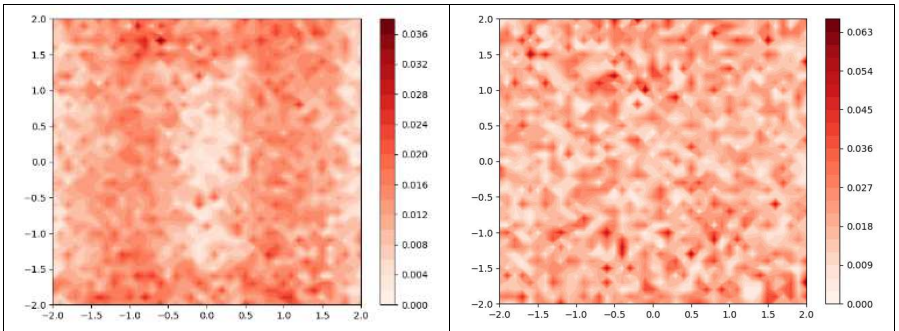


*Figure 35 With 3% noise*

*Figure 36 With 10% noise*

Figure 35Figure 36 show the results. Both figures are differently scaled. Their "redness" is not comparable. The maximum of the right plot is almost twice as big as the left. In Figure 35, the pattern from Figure 31 is blurry but still recognizable. In figure, there is no pattern at all – just noise.

# 6 Discussion

## 6.1 Possible features

Of course, this thesis' implementation cannot at all compete with a whole team like Pupil Labs has. However, the implementation is a good starting point for implementing a lot of interesting features that could be increasing the accuracy directly or indirectly.

- Since there is no ground truth data in a real-world setting, the tracking accuracy is not directly available. Two different accuracy measurements could be implemented. One for how well the eyeball center and one for how each single gaze is tracked.
- When section 2 would be completely implemented the 3D gaze could be tracked. The tracked longitude, latitude and pupil radius could then again be projected with the implementation of section 3.1 and then tracked again. This loop could be repeated as often as wanted to see if and how the data converges.
- During the process of finding the eyeball center (see Figure 10), machine learning methods like clustering could be used to detect head slippage. After head slippage a new eyeball center would be initialized. Each cluster would represent an eyeball position.
- The eye tracking camera is built in manually in the VR googles. So, the spatial relation between the camera and the VR googles' screen is not exactly known as it would be the requirement for absolute eye movements tracking without further manual calibration. If they eye tracking camera would be firmly installed by the VR googles manufacturer, all googles of the same model would have the same spatial relation between camera and display. Absolute eye movement tracking without further manual calibration could be possible. Future generation of VR googles might be coming with built in eye tracking.

## 6.2 How to address software projects

It is easy to lose the overview in software projects. It is easy to lose the overview in software projects and make things rather worse rather than better. Of course, every software project is different but there are a few things and best practices that almost always pay off when they are used.

First and quite obvious, it is very important to work precisely. Using the wrong OpenCV documentation version lead to implement of a wrong ellipse interface. It is easy to solve such problems but it can be very tedious to find such bugs because the unit tests were designed with the wrong documentation in mind. Of course, one cannot be 100% attentive all day long. The best practice here is to automate as much as possible and be especially aware when using manual solutions. For the OpenCV example, it means that the documentation should be whenever possible be looked up inside the IDE and not online because the IDE always uses the right documentation version.

Before starting to code, the configurational set up should be clear. What does that exactly mean? The most crucial configurational set up of this thesis was the coordinate system (details can be found in section 1.3). The configurational set up of this thesis coordinate system was taken from the ellipse reprojection (Safaee-Rad, 1992). Unfortunately, the projection *projection.py* was implemented with an arbitrary coordinate system configuration beforehand. Because of these different configurations, some modules had to be implemented twice – for the projection as well as reprojection which leads to accidental implementation of two different focal lengths. The best practice here is to not only understand the thing to be implemented at a conceptual level but to consider the configurational aspects, too. Furthermore, it should be considered how easy the configuration could be changed. The harder the configuration is to change the better it should be thought through before the actual implementation process.

Every at least slightly complicated function should be tested in an isolated manner. The conclusion from section 3.6.1 where different T1 were used for the positive and negative conclusion only took place because it was not clear were exactly the mistake was. A unit test would have quickly revealed

that the mistake was in the projection of a 3D line to 2D. If a function is not testable in an isolated manner, the algorithmic and data structure should be changed such that function is single module afterwards.

At last, there is question that cannot be totally answered with a best practice: should the proposed solution be used or might another approach be better? The approaches in the alternative approaches section are all useless in the end. It can be said that an alternative solution is not easily and quickly found for bigger problems such as finding the elliptic reprojection. For smaller detail questions, however, the bottom line is mixed: The cone transformation pipeline could be simplified (see section 3.2.1), whereas the rotation of the cone would have made less problems if implemented with the discriminating cubic as done by Safaee-Rad[17] (see section 3.5). The question whether the proposed solution or another approach should be used is very topic specific. The only really reliable source here might be the experience one has in the topic of question.

# Bibliography


Altomare, L., 2015. Eye Tracking Detects Disconjugate Eye Movements Associated with Structural Traumatic Brain Injury and Concussion. *Journal of NeurotraumaVol. 32, No. 8*, 07 04, pp. 548 - 556.

Anon., 2014. *Auch Fifa beschließt Drei-Minuten-Pause.* [Online]
Available at: https://www.spiegel.de/sport/fussball/gehirnerschuetterungen-auch-fifa-beschliesst-drei-minuten-pause-a-993252.html

Anon., kein Datum *Kreisgleichungen.* [Online]
Available at: https://www.massmatics.de/merkzettel/#!700:Kreisgleichung

Antoniades, C. & Kennard, C., 2014. Ocular motor abnormalities in neurodegenerative disorders. *Eye*, 29 02, pp. 200-207.

House, D. H., kein Datum *University Clemson: Computer Graphics Course, Implicit and Parametric Surfaces.* [Online]
Available at: https://people.cs.clemson.edu/~dhouse/courses/405/notes/implicit-parametric.pdf

mindQ GmbH, 2018. *eyeTrax.* [Online]
Available at: eyetrax.de

R. Safaee-Rad, I. T. e. a., 1992. *Three-dimensional location estimation of circular feature.* s.l., IEEE.

Schäfers, A.-K., 2019. *Gewinner des Digitalen Gesundheitspreises 2019 von Novartis und Sandoz Deutschland/Hexal.* [Online]
Available at: https://www.novartis.de/news/media-releases/die-gewinner-des-digitalen-gesundheitspreises-2019-von-novartis-und-sandoz

Schneider, A., kein Datum *Mathebibel.* [Online]
Available at: https://www.mathebibel.de/hauptachsentransformation

Swirski, L. & Dodgson, N., 2013. *A fully-automatic, temporal approach to single camera, glint-free 3D eye model fitting.* Lund, Sweden, s.n.


[17] Safaee-Rad only suggest diagonalization as an alternative for rotating the cone to its major axis. He himself uses the discriminantic cubic.

# Acknowledgements

I would like to thank the following persons for helping me with this thesis:

Prof. Dr. Peter König for supervision and feedback.

Dr. Matthias Temmen for supervision, feedback and answering all kinds of question and especially for consulting me whenever I was stuck.

Christopher Gundler for introducing me to mindQ and for introducing me to professional software engineering practices.

Paul Liebenow for showing me how mathematicians' approach difficult problems.

Philip Fürste and Michel Nöring for reading this thesis and for grammatical corrections.

Thank you all very much. This thesis would not have been possible without you.


# Appendix

## 6.3 UML class diagram

*Diagram 1 The projection.py module*

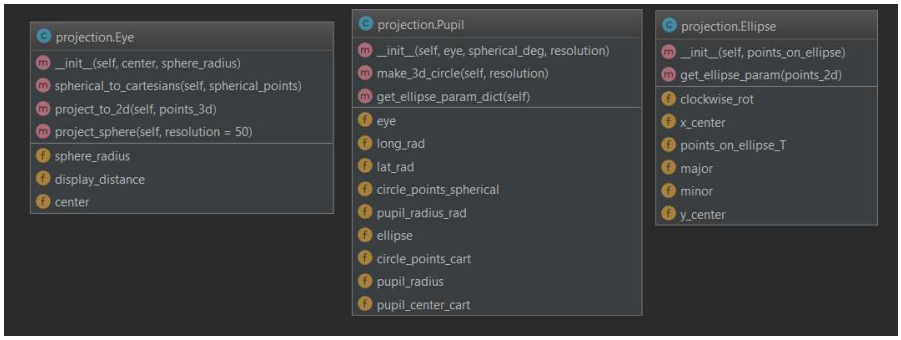
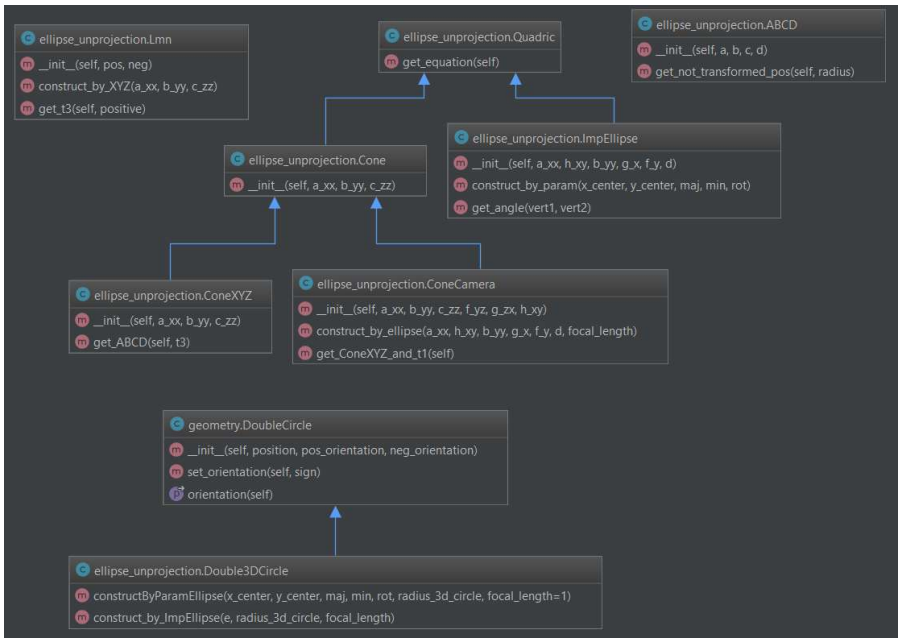


*Diagram 2 The detailed class diagram for the ellipse reprojection*




## 6.4 Code Listings

# genetic_reprojection.py

*The beginning implementation of the genetic algorithm approach. The last lines calculate the coordinates of the shadow hitting the eyeball. This approach was eventually rejected because it was thought that a good evaluation function cannot be found.*

```
import numpy as np
from math import sqrt
import projection


 class shadow_mapped_eye(projection.Eye):
    def __init__(self):
        '''
            The eye on which the ellipses are shadow mapped on.
        '''
        #TODO Set random center later or take some heuristic
        projection.Eye.__init__(self)


 class shadow_mapped_pupil():
    def __init__(self,shadow_mapped_eye, ellipse):
        '''
        Take ellipse as input for shadowmapping.
        '''
        #TODO: np.nditer(a, flags=['external_loop'], order='F'): for performance
        #Achtung traversiert!
        points = ellipse.points_on_ellipse_T
        res = points.shape[0]
        line_length_array = np.empty(res)

         a = shadow_mapped_eye.center[0]
```

```
        b = shadow_mapped_eye.center[1]
        c = shadow_mapped_eye.center[2]
        r = shadow_mapped_eye.sphere_radius

         for i in range(res):
            point = points[i]
            x = point[0]
            # Is display size
            y = shadow_mapped_eye.display_distance
            z = point[1]
            #TODO CHECK IF REALLY ELLIPSE points
            u = (-sqrt((-2*a*x - 2*b*y - 2*c*z)**2 - 4*(x**2 + y**2 + z**2)*(a**2 + b**2 +
c**2 - r**2)) + 2*a*x + 2*b*y + 2*c*z)/(2*(x**2 + y**2 + z**2))
            line_length_array[i] = u




 #For Execution
s_e = shadow_mapped_eye()
p = projection.Pupil(s_e, [0,0,0.2])
s_p = shadow_mapped_pupil(s_e, p.ellipse)
```

# geometry.py

*DoubleCirlce is implemented such that when the right circle orientation is know it can be set with the function set_orientation (would be implemented later). The function orientation comes with the @property operator. This makes orientation, although a function, accessible as an object attribute.*

```
"""
Module for all geometry classes and methods that are used in multiple modules
"""
import numpy as np


class DoubleCircle:
    def __init__(self, position, pos_orientation, neg_orientation):
        self.position = position
        self.pos_orientation = pos_orientation
        self.neg_orientation = neg_orientation
        self.sign = None

    def set_orientation(self, sign):
        """
        Sets the orientation in order to solve two-circle ambiguity
        :param sign: True for positive, False for negative
        """
        if self.sign is not None:
            raise PermissionError("Sign can only be set once")
        self.sign = sign

    @property
    def orientation(self):
        if self.sign is None:
            raise ValueError("True orientation is not set yet")
        if self.sign:
            return self.pos_orientation
        else:
            return self.neg_orientation


class ParametricEllipse:
    def __init__(self, x_center, y_center, maj, min, rot):
        if maj < min:
            raise ValueError("Major axis needs to be bigger than minor")
        if not (0 <= rot <= 180):
            raise ValueError("Not more than 180 degrees rotation possible")
        self.x_center = x_center
        self.y_center = y_center
        self.maj = maj
        self.min = min
        self.rot = rot


def project_to_2d(point_3d, focal_length=1, return_2d=True):
    """
    projected plane is xy
    :param point_3d: np.array([x,y,z])
    :param focal_length:
    :param return_2d: if True than 2d points will be returned,
    else as 3d elements on plane
    :return:
    """
    flat_x = focal_length*point_3d[0]/point_3d[2]
    flat_y = focal_length*point_3d[1]/point_3d[2]
    if return_2d:
        return np.array([flat_x, flat_y])
    else:
        return np.array([flat_x, flat_y, focal_length])


class Line:
    def __init__(self, pos, orientation):
        self.position = pos
        # Normalize orientation to length 1
```

```
        self.orientation = orientation / np.linalg.norm(orientation)


def intersecting_lines(line_list):
    """
    Intersecting with least squares method
    :param line_list: list with line objects
    :return: projected eye center as 2d np array
    """
    dim = np.size(line_list[0].position)

    first_multiplicand = np.zeros(shape=(dim, dim))
    second_multiplicand = np.zeros(shape=(dim))

    for l in line_list:
        summand = np.eye(dim) - np.outer(l.orientation, l.orientation)

        first_multiplicand = first_multiplicand + summand
        second_multiplicand = second_multiplicand + np.dot(summand, l.position)

    first_multiplicand = np.linalg.inv(first_multiplicand)
    return np.dot(first_multiplicand, second_multiplicand)
```

# projection.py

```
import numpy as np
import math
import cv2 as cv

"""
xz plane ist standard bis jetzt
"""


class Eye:
    def __init__(self, center, sphere_radius):
        '''
        Angaben entprechen cm.
        :param center:
        :param sphere_radius:
        '''
        if center[1] + sphere_radius > -1:
            raise ValueError("Eyeball muss weiter nach hinten auf der y-Achse verschoben
werden, sonst nicht von Kamera erfasst")
        else:
            self.center   = center
            self.sphere_radius = sphere_radius
            self.display_distance = round(self.center[1]+self.sphere_radius)/2 #Should be neg-
ative!

    def spherical_to_cartesians(self, spherical_points):
        """
             Neutral pos:: (long, lat, spherical_rad): (0,0,1) -> (0,1,0) in cartesian
             Spherical points are in regards to eye center which is considered for the recal-
culation
            :param spherical_points: numpy array with shape (points size, 3)
            :return: cartesian numpy
         """

        if spherical_points.shape[0] == 2:
            radius = self.sphere_radius
        else:
            radius = spherical_points[2]

        hor = spherical_points[0]
        vert = spherical_points[1]

        x = np.sin(hor) * np.cos(vert) * radius + self.center[0]
        y = np.cos(hor) * np.cos(vert) * radius + self.center[1]
        z = np.sin(vert) * radius + self.center[2]
        return np.array([x, y, z])

    def project_to_2d(self, points_3d):
        if self.display_distance > 0:
            raise ValueError("Should be negative")
        '''
        Takes roughly half the distance of sphere surface and origin
        :param points_3d: cartesian points numpy array with shape(amount of points, 3)
        :return: 2d numpy array with shape(amount of points, 2)
        '''
        near = self.display_distance
        flat_x = points_3d[0]*near/points_3d[1]
        flat_z = points_3d[2]*near/points_3d[1]

        return np.array([flat_x, flat_z])

    def project_sphere(self, resolution = 50):
        raise NotImplementedError('Need to be restructured for new data structure')

        # draw sphere
        long = np.arange(0,math.pi*2,math.pi*2/resolution)
        lat = np.arange(0,math.pi*2,math.pi*2/resolution)

        grid = np.empty((long.size * lat.size,2))
        for i in range(0,long.size):
            for j in range(0, lat.size):
                grid[i*long.size + j][0] = long[i]
                grid[i * long.size + j][1] = lat[j]
```

```
        sphere = self.rads_to_cartesians(grid)
        return self.project_to_2d(sphere)


class Pupil:
    def __init__(self, eye, spherical_deg, resolution):
        """
        :param eye:
        :param spherical_deg: [long, lat, pupil_radius]
        :param resolution:
        """
        self.long_rad = math.radians(spherical_deg[0])
        self.lat_rad = math.radians(spherical_deg[1])
        self.pupil_radius = spherical_deg[2]
        self.eye = eye
        self.circle_points_spherical = self.make_3d_circle(resolution)
        self.circle_points_cart = self.eye.spherical_to_cartesians(self.circle_points_spheri-
cal)
        self.pupil_center_cart = self.eye.spherical_to_cartesians(np.array([self.long_rad,
self.lat_rad]))
        self.ellipse = Ellipse(self.eye.project_to_2d(self.circle_points_cart))

    def make_3d_circle(self, resolution):
        '''
        :return: Cartesian 3D coordinates from pupil
        '''
        self.pupil_radius_rad = np.arcsin(self.pupil_radius/self.eye.sphere_radius)

        u = np.arange(0, math.pi*2, math.pi*2/resolution)
        long = np.cos(u)*self.pupil_radius_rad + self.long_rad
        lat = np.sin(u)*self.pupil_radius_rad + self.lat_rad

        return np.array([long, lat])

    def get_ellipse_param_dict(self):
        return {"x_center": self.ellipse.x_center,
                "y_center": self.ellipse.y_center,
                "maj": self.ellipse.major,
                "min": self.ellipse.minor,
                "rot": self.ellipse.clockwise_rot}


class Ellipse:
    def __init__(self, points_on_ellipse):
        self.points_on_ellipse_T = np.transpose(points_on_ellipse)
        ellipse_cv = self.get_ellipse_param(self.points_on_ellipse_T)
        #TODO check if interface is correct
        self.x_center = ellipse_cv[0][0]
        self.y_center = ellipse_cv[0][1]
        self.major  = ellipse_cv[1][1]
        self.minor  = ellipse_cv[1][0]
        #Rotate with 90 degrees because the axis are changed
        #between projection and unprojection
        self.clockwise_rot = (ellipse_cv[2]+90) % 180

    @staticmethod
    def get_ellipse_param(points_2d):
        '''
        ellipse_cv has following parameters:
        center(x,y), (major_axis, minor_axis), clockwise_rotation, 0, 360, color=255,
line_thickness
        :param points_2d TRAVERSED
        :return:
        '''
        if points_2d.shape[0] == 3:
            raise TypeError("Nimmt nur traversierte np arrays an")
        #TODO check if interface is correct
        ellipse_cv = cv.fitEllipse(points_2d.astype(np.float32))
        return ellipse_cv
```

# execution.py

```
import projection
import numpy as np
import eye_tracking
import matplotlib.pyplot as plt


symmetric_30 = [[-30.0, -30.0, 0.2], [-30.0, 0.0, 0.2], [-30.0, 30.0, 0.2], [0.0, -30.0, 0.2],
[0.0, 30.0, 0.2], [30.0, -30.0, 0.2], [30.0, 0.0, 0.2], [30.0, 30.0, 0.2]]
almost_symmetric_30 = [[-30.0, -30.0, 0.2], [-30.0, 0.1, 0.2], [-30.0, 30.0, 0.2], [0.1, -
30.0, 0.2], [0.1, 30.0, 0.2], [30.0, -30.0, 0.2], [30.0, 0.1, 0.2], [30.0, 30.0, 0.2]]

####PARAMETERS################
EYE_CENTER = np.array([0.0, -5.0, 0.0])
SPHERE_RADIUS = 1.2
RESOLUTION = 100
RADIUS_3D_CIRCLE = 4
PUPIL_PARAM_LIST = almost_symmetric_30
FOCAL_LENGTH = 1

if __name__ == "__main__":
    #Projection
    eye = projection.Eye(EYE_CENTER, SPHERE_RADIUS)
    ellipse_param_list = []
    for i in range(len(PUPIL_PARAM_LIST)):
        p = projection.Pupil(eye, PUPIL_PARAM_LIST[i], RESOLUTION)
        ellipse_param_list.append(p.get_ellipse_param_dict())

    #Unprojection
    results = eye_tracking.run(ellipse_param_list, RADIUS_3D_CIRCLE, FOCAL_LENGTH)

    real_projected_center = [EYE_CENTER[0]/-EYE_CENTER[1], EYE_CENTER[2]/-EYE_CENTER[1]]

    distance = np.linalg.norm(results - real_projected_center)
    print(distance)
    plt.plot(real_projected_center[0], real_projected_center[1], marker='v')
    plt.show()

def gaze_maker(long_list, lat_list, radius):
    '''
    Combines every option of long and lat that is given
    :param long_list: list of spherical
    :param lat_list: list of spherical
    :param radius: single value
    :return: numpy array
    '''
    output = []
    for long in long_list:
        for lat in lat_list:
            output.append([long, lat, radius])
    return output
```

# tests.test_eye_tracking.py

```
from unittest import TestCase


class TestIntersecting_lines(TestCase):
    def test_intersecting_lines(self):
        from geometry import Line
        from geometry import intersecting_lines
        import numpy as np

        l1 = Line([0, 1], [.5, .5])
        l2 = Line([0, -1], [.5, -.5])

        intersection = intersecting_lines([l1, l2])

        np.testing.assert_almost_equal([-1, 0], intersection)
```

# tests.test_projection.py

```
from unittest import TestCase


class TestProjeciton(TestCase):
    def test_get_ellipse_param_dict(self):
        places = 3

        from projection import Eye, Pupil
        import numpy as np
        e = Eye(np.array([0.0, -5.0, 0.0]), 1.2)

        p = Pupil(e, [0.0, 0.0, 0.2], resolution=50).get_ellipse_param_dict()
        self.assertAlmostEqual(0, p["x_center"])
        self.assertAlmostEqual(0, p["y_center"])
        self.assertAlmostEqual(p["maj"], p["min"], places)

        p = Pupil(e, [30.0, 30.0, 0.2], resolution=50).get_ellipse_param_dict()
        self.assertGreater(p["x_center"], 0)
        self.assertGreater(p["y_center"], 0)
        self.assertAlmostEqual(p["rot"], 135, delta=15)

        p = Pupil(e, [-30.0, 30.0, 0.2], resolution=50).get_ellipse_param_dict()
        self.assertGreater(0, p["x_center"])
        self.assertGreater(p["y_center"], 0)
        self.assertAlmostEqual(p["rot"], 45, delta=15)

        p = Pupil(e, [0.0, 30.0, 0.2], resolution=50).get_ellipse_param_dict()
        self.assertAlmostEqual(0, p["rot"])
        self.assertAlmostEqual(0, p["x_center"])
        self.assertGreater(p["y_center"], 0)

        p = Pupil(e, [30.0, 0.0,  0.2], resolution=50).get_ellipse_param_dict()
        self.assertAlmostEqual(90.0, p["rot"])
        self.assertAlmostEqual(0, p["y_center"])
        self.assertGreater(p["x_center"], 0)
```

# tests.test_unprojection.py

```
from unittest import TestCase
from tests import test_transformation as trans
import numpy as np

places = 2


class TestImpEllipse(TestCase):
    def test_construct_by_param(self):
        from ellipse_unprojection import ImpEllipse
        e_param = ImpEllipse.construct_by_param(-1.53333, 1.1333, 3.218, 2.49266, 135)
        e_imp = ImpEllipse(1, 0.5, 1, 2.5, -1.5, -5)
        self.assertAlmostEqual(e_param.a_xx, e_imp.a_xx, places)
        self.assertAlmostEqual(e_param.h_xy, e_imp.h_xy, places)
        self.assertAlmostEqual(e_param.b_yy, e_imp.b_yy, places)
        self.assertAlmostEqual(e_param.g_x, e_imp.g_x, places)
        self.assertAlmostEqual(e_param.f_y, e_imp.f_y, places)
        self.assertAlmostEqual(e_param.d, e_imp.d, places)


class TestUnprojection(TestCase):
    from ellipse_unprojection import ImpEllipse
    from ellipse_unprojection import Double3DCircle

    # Results from Safaee-Rad p.632
    e_imp = ImpEllipse(204.024, -102.452, 225.000, -127.567, -177.45, 66.976)
    double3d_from = Double3DCircle.construct_by_ImpEllipse(e_imp, 4, focal_length=1)

    def test_construct_by_ImpEllipse(self, double3d_from = double3d_from):
        pos_true = np.array([11.830, 13.660, 27.811])
        orientation_true = np.array([-0.5, 0, -0.866025])
        decimal=2
        np.testing.assert_almost_equal(double3d_from.position, pos_true, decimal=decimal)
        np.testing.assert_almost_equal(double3d_from.pos_orientation, orientation_true, deci-
mal=decimal)

    def test_orientation_projection(self, double3d_from = double3d_from):
        from eye_tracking import ProjectedPupil
        ProjectedPupil.construct_by_Double3DCircle(double3d_from, focal_length=1)
```

# README.md

```
# **Self calibrating eye tracking**
Usually eye-tracking devices need to be manually calibrated by the user. This one
calibrates itself without the user noticing at all. For a detailed description of
the underlying concept have a look at
[this paper](http://2013.petmei.org/wp-content/uploads/2013/09/petmei2013_session2_3.pdf).
I did this project for my bachelor thesis which is part of
[eyeTrax](https://www.eyetrax.de/):
an eye tracking system for diagnosing light concussions based on occulomotoric deficits.

# Installation

Create and activate the conda environment from the `env_linux_64.yml` file: <br>
`conda env create -f env_linux_64.yml` <br>
`conda activate ba_env` <br>
NOTE: As the filename already says, it only works with Linux 64bit

# Usage
Run `execute.py` to simulate and eye tracking session. For setting the parameters of the
simulation have a look at the documentation of `execution.py`.
It should output the results as a Matplotlib figure.
### Using own data
Alternatively to simulating, you can use your own recorded tracking data. This is an
experimental feature yet. The recorded pupils need to be provided as 2D ellipse.
All pupils are saved in a python list and each pupil is a dictionary in the following
form. <br>
`{"x_center": self.ellipse.x_center,
                "y_center": self.ellipse.y_center,
                "maj": self.ellipse.major,
                "min": self.ellipse.minor,
"rot": self.ellipse.clockwise_rot}` <br>
The rotation is measured in degrees. You need to replace the `ellipse_param_list` with
your own in the `executin.py` file.
```

---

*Declaration of Authorship*

---

I hereby certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other university.

_____

Marko Duda

_____

city, date