Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Espen Simon Liavik

Combining laser triangulation with rotary measurements to generate a positional basis for a welding robot

Master's thesis in Industrial Cybernetics
Supervisor: Ole Morten Aamo
June 2019

NTNU
Norwegian University of Science and Technology

Espen Simon Liavik

Combining laser triangulation with rotary measurements to generate a positional basis for a welding robot

Master's thesis in Industrial Cybernetics
Supervisor: Ole Morten Aamo
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Kunnskap for en bedre verden

Abstract

The company WellConnection Mongstad reports that a time-consuming welding process of rotating drill-pipes causes a bottleneck within their production line. In order to remove this bottleneck, the company desires to look in to the possibility of automating this process with the use of a welding robot. However, if a robot is to perform such a task automatically, it needs to be provided information regarding the geometry of the drill-pipes. Consequently, the focus of this thesis was to explore a method which can be used to scan the rotating drill-pipes which supplies the data needed to calculate a positional reference.

The approach of the thesis has been to combine computer vision with a measurement of the pipe's orientation, such that the curvature of the pipe is known for any given orientation. With regards to computer vision, this thesis specifically uses laser triangulation as a method for capturing the curvature. This method composes a laser-projector and a camera, which is a vastly used method for expressing an objects shape in three dimensional coordinates.

As the relationship between the laser projector and the camera is essential in laser triangulation, the thesis has also explored a method for identifying this relationship. This was achieved by generating a pointcloud of points located at the plane of the laser, from where the laserplane was estimated by using different plane fitting methods. As the planes were estimated, they were tested in their accuracy by calculating known dimensions of an object. Moreover, the best accuracy was achieved when weights was applied to the points which corresponded to their positional uncertainty.

In order to provide an orientational measurement a rotary encoder was connected to an Arduino using an analog-digital converter. As the measurement of orientation was combined with the reading from the laser triangulation, a 3D-printed cylindrical object was used as a test-subject for the scanning process. The object was rotated in both a centric and non-centric fashion, from where the data of the scan was presented in terms of point-clouds. The final representation illustrates the curvature of the object as a function of its orientation. This representation was used in order to demonstrate how the welding robots path could be estimated. When applied to one of the objects grooves, the estimated path demonstrated a high degree of correspondence to the rotational offset of the object.

Sammendrag

Bedriften WellConnection Monstad rapporterer at en tidskrevende sveise-prosess av roterende borerør forårsaker en flaskehals i produksjonslinjen deres. Etter et ønske om å fjerne denne flaskehalsen, vil bedriften undersøke muligheten for å få denne prosessen automatisert ved å benytte en sveiserobot. Dersom en robot skal kunne utføre en slik oppgave på automatisert vis, må den kunne forsynes med informasjon relatert til borerørets geometri. Av denne grunn er fokuset til denne oppgaven å utforske en metode som kan bli brukt til å skanne de roterende borerørene for å generere den nødvendige dataen som videre kan bli brukt for å skape en posisjonsreferanse.

Tilnærmingen til denne oppgaven har vært å kombinere datasyn med en måling av borerørets orientasjon, slik at rørets kurvatur er kjent for enhver orientasjon. Innen datasyn har denne oppgaven brukt lasertriangulering som en metode for å kunne måle rørets kurvatur. Denne metoden består av en laserprojektor og et kamera, og er en ofte brukt metode for å utrykke formen av et objekt i tredimensjonale koordinater.

Ettersom forholdet mellom laserprojektoren og kameraet er meget viktig innen lasertriangulering, har oppgaven også tatt for seg en metode som identifiserer dette forholdet. Dette ble gjort ved å generere en punktsky av punkter lokalisert i laserens plan, hvor laserplanet deretter ble estimert ved bruk av forskjellige planestimerings-metoder. Resultatene fra de forskjellige metodene ble deretter testet i sin nøyaktighet ved å måle kjente dimensjoner langs et objekt. Det mest nøyaktige planet ble estimert ved å bruke vekting av punktene i punktskyen, hvor vektene ble kalkulert etter hvor mye usikkerhet som var knyttet til de forskjellige punktene.

For å få tilgang til en orientasjonmåling ble det brukt en rotasjonsenkoder tilkoblet en Arduino via en analog-digital konverter. Når denne målingen ble kombinert med målingene fra lasertrianguleringen, ble et 3D-printet sylindrisk objekt skannet. Objektet ble rotert både med og uten kast, hvor den resulterende dataen videre ble illustrert ved å generere punktskyer. Den endelige representasjonen demonstrerte objektets kurvatur som en funksjon av orientasjon, og ble brukt til å illustrere hvordan robotens bane kunne bli kalkulert. Dette ble gjort for en av objektets seksjoner, og viste et høyt samsvar med objektets planlagte kast.

Preface

This thesis is written during the last semester of my two year master program at the Department of Engineering Cybernetics at NTNU. Although I feel that the results have been restricted by the inability to perform on-site experiments, I am content with the results of this thesis.

I would like to thank everyone who has contributed to the thesis. A special thanks to my supervisor Ole Morten Aamo at IKT and co-supervisor Lars Tingelstad at MTP for guidance and assistance throughout the process of the thesis. Next, I would like to thank co-student Trym D. Hauglund for a mutually beneficial cooperation which eased the workload for us both. I would also like to thank the department of mechanical and industrial engineering at NTNU for providing the camera and laser which made the experiments in this thesis possible. Additionally I would express my gratitude to Roman Handke at Automation Technology for providing a standalone Python-package which included all the necessary functions which allowed communication with the camera.

Lastly, I would like to thank my fiancée Brigitte for the unconditional support and encouragement during my many years of education. Without you, all these years would have felt even longer.

Trondheim, June 3, 2019

Espen Simon Liavik

Contents

List of Figures

1. Introduction

1.1 Background

WellConnection Mongstad is a company located in Austrheim who specializes in inspection and repair of equipment related to offshore oil-production. A part of their speciality is to inspect and repair drill-pipes which has been used in deep-sea drilling. Due to impurities in the metal, some drill-pipes experience accumulation of pores in the walls after being used in the drilling process. As these pores weaken the integrity of the metal, the surface of the pipes are machined down such that the pores are removed.

As the metal containing the pores is removed, the cross-section of the pipe is reduced and needs to be fill-welded in order to achieve the original dimension. WellConnection has stated that the welding is performed using MIG arc welding, where the welding is preferred to be located at the top section of the pipes, as this increases the quality of the weld.

The current method for this operation involves the drill-pipes being mounted on a rotary chuck and rotated as a worker welds a series strings around the circumference of the machined section of the pipe. This process is stated to be a time-consuming task which requires a large amount of man-hours. Consequently, the company desires to automate this welding process with the use of a welding robot. However, in order to automate such a process the robot requires geometric information of the drill-pipe that can be used as positional reference for its welding gun.

Furthermore, some of the drill-pipes are reportedly rotating in a non-centric fashion due to wear and tear. Consequently, the non-centric rotation should be taken into consideration if a robot is to be supplemented with a positional reference.

1.1.1 Problem formulation and objectives

In order to generate a basis which can be used for calculating a path for the robots welding-gun, this thesis will through experiments investigate if laser triangulation combined with an angular measurement is a suited method. Laser triangulation is a common method within the field of 3D vision which is used to provide three-dimensional readings of an objects shape [1]. As a laserline is projected onto the object, the location of the laser is registered in the cameras image, which then can be expressed in three dimensional coordinates. As the pipe is rotated, the orientation is to be measured such that each reading of the pipes curvature can be directly related to the measured angle. However, in order to achieve reliable results from the experiments – there are two additional key elements which needs to be addressed. The accuracy of laser triangulation is heavily dependent on an accurate estimation of the translational and orientational relationship between the laser projector and camera. As a result, a part of the thesis will be to identify this relationship and quantify its accuracy to determine if it is sufficient for the use of a welding robot. In addition, as the proposed system relies on an angular reading of the scanned object, a circuit which provides the reading will be developed such that this reading is available during the scanning of the object.

The main objectives of this thesis can therefore be stated with the following list.

- Ensure the availability of an orientational measurement

- Perform a laserplane calibration and study the accuracy of the estimated plane
- Perform a scan of an object which combines both the readings of curvature and orientation
- Process the data from the scan such that it can be used as a basis for positional reference

1.2 Approach

This thesis is written with a quantitative approach where most of the results are found by experiments and processing of the acquired data. As the purpose of the thesis is to deliver a basis for positional input to a welding robot, accuracy of the system has been weighted more opposed to other factors such as computing speed and simplicity. The work presented in this thesis began with acquiring information regarding the subject using both scientific literature found online and relevant books within the field of computer vision. When possible, search engines such as IEEE and ScienceDirect was used in order to refer to peer-reviewed articles which is regarded as a more reliable source of material. Additional information was also found from other sources, such as manufacturers. However, as information directly from the manufacturers can be biased, this information was treated with care. Moreover, since most of the results are derived from physical experiments, the results are believed to be more resilient to biased information.

Initially the experiments was planned to take place on-site at WellConnection, however due to the time required to get the system operative – this was not feasible. Instead it was decided that the actual scanning would be performed using a 3D-printed shape. However, since the staged experiments did not reflect all the on-site conditions the results may not be directly transferable to the scanning of a drill-pipe.

In addition, since co-student Trym D. Hauglund was writing a similar thesis regarding laser triangulation, Lars Tingelstad advised a cooperation in order to reduce the individual workload. As most of this workload was based on writing the appropriate programs, some of the code used in this thesis is written by Hauglund. Consequently, Hauglund is directly credited as his code is used in the different sections of the thesis.

Throughout the thesis there are written various functions and programs which allows treatment of the various data. A majority of this code is written in the high-level programming language Python. This is one of the most popular programming languages [2], and allows the use of several libraries which enables data-processing on a scientific level. The following libraries was utilized throughout the thesis and allowed visualization and further processing of the data.

- Numpy : Simplifies data-array handling
- Matplotlib : For visualization of plots and figures
- PPTK : For processing dense point-clouds
- SciPy : Allows scientific data processing such as optimizing and curve-fitting
- OpenCV : For gaining access to functions related to computer vision

1.3 Structure of the report

In chapter 2 the various theory which is used throughout the thesis is given an introduction. This chapter is arranged in such a way that the elementary subjects are introduced first, which is then used in the introduction of more intricate subjects. The next part of the thesis, chapter 3, introduces the equipment and methods used in the experiments. The topics of this chapter includes an explanation of how the camera operates, and how it was mounted on a rail with the laser. In chapter 4 the performed experiments are given a description, in addition to the results from the different experiments. The results are then discussed in chapter 5 from which a conclusion is drawn and further work suggested in chapter 6.

Additionally, the appendix contains the code which is generated throughout the thesis and will be referred to as the different code becomes relevant. Files which is not attached in the appendix can be found in the digital appendix.

2. Preliminaries

This chapter describes the theory which in used in computing, interpretation and processing of the data of this thesis. Elementary subjects such as rotation matrices and translational vectors are first explained, which is then used in explaining homogeneous transformations. Additional subjects such as robot welding, laser triangulation and camera calibration are also given an introduction.

2.1 Rotation matrices

A rotation matrix is a representation which describes a rotation with respect to a given frame. Considering the simple rotations about the x, y and z axes – the rotation matrices $\mathbf{R} \in \mathbb{R}^{3x3}$ can be represented as following [3].

$$\mathbf{R}_x(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \tag{2.1}$$

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \tag{2.2}$$

$$\mathbf{R}_z(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

A rotation from an arbitrary frame a to frame b can be expressed by combining the different simple rotations, such that the desired orientation is achieved. A common way to describe such a rotation is with the use of ZYZ Euler angles[3]. The rotation from frame a to frame b can be written as

$$\mathbf{R}_b^a = \mathbf{R}_{ZYZ} = \mathbf{R}_z(\psi)\mathbf{R}_y(\theta)\mathbf{R}_z(\phi) \tag{2.4}$$

By carrying out the matrix multiplication, the resulting rotation matrix is,

$$\mathbf{R}_b^a = \begin{bmatrix} c_\psi c_\theta c_\phi - s_\psi s_\psi & -c_\psi c_\theta s\phi - s_\psi c_\phi & c_\psi s_\theta \\ s_\psi c_\theta c_\phi + c_\psi s_\psi & -s_\psi c_\theta s\phi - c_\psi c_\phi & s_\psi s_\theta \\ -s_\theta c_\phi & s_\theta s_\phi & c_\theta \end{bmatrix} \tag{2.5}$$

Furthermore, the column-vectors of the matrix $\mathbf{R}_b^a$ can also be interpreted as the directions of the axes of frame b with respect to frame a[4].

$$\mathbf{R}_b^a = \begin{bmatrix} \overline{\mathbf{x}} & \overline{\mathbf{y}} & \overline{\mathbf{z}} \end{bmatrix} \tag{2.6}$$

2.1.1 Properties of rotational matrices

By definition, a 3x3 matrix $\mathbf{R}$ is only a rotation matrix if and only if it is an element of the special orthogonal group $SO(3)$ [3]. Consequently, there are three conditions which needs to be fulfilled in order for the matrix $\mathbf{R}$ to be a rotation matrix.

$$SO(3) = \{\mathbf{R} | \mathbf{R} \in \mathbb{R}^{3x3}, \quad \mathbf{R}^T\mathbf{R} = I, \text{and} \quad \det \mathbf{R} = 1\} \tag{2.7}$$

2.1.2 Angle-axis parameters

An arbitrary rotation can also be stated on angle-axis representation. This representation is based on a rotation axis $\mathbf{k}$ and the angle of rotation θ [3]. By applying Rodrigues formula, this representation can be presented as a rotation matrix as shown below.

$$\mathbf{R} = \mathbf{I} + \sin\theta\mathbf{k}^{\times} + (1 - \cos\theta)\mathbf{k}^{\times}\mathbf{k}^{\times} \quad \text{, where} \quad \mathbf{k}^{\times} = \begin{bmatrix} 0 & -k_z & k_y \\ k_z & 0 & -k_x \\ -k_y & k_x & 0 \end{bmatrix}$$

Furthermore, the angle-axis representation of an arbitrary rotation matrix $\mathbf{R}$ can be derived by the following method [4]. If the 3x3 rotation matrix is given as,

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

The angle of rotation can be calculated as,

$$\theta = \cos^{-1}\left(\frac{r_{11} + r_{22} + r_{33} - 1}{2}\right) \tag{2.8}$$

And the axis of rotation can be computed as

$$\mathbf{k} = \frac{1}{2\sin\theta}\begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix} \quad \text{, for} \sin\theta \neq 0 \tag{2.9}$$

2.2 Translational vectors

In order to represent relative position between two points in space, the translational vector is commonly used. The displacement between a frame a and b can be described by the following vector.

$$\mathbf{r}_{ab}^{a} = \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} \tag{2.10}$$

The superscript describes which frame the vector is expressed with respect to, while the subscript defines the direction. The vector (2.10) is then the vector from frame a to b, with respect to frame a. Consequently, the following statement hold.

$$\mathbf{r}_{ab}^{a} = -\mathbf{r}_{ba}^{a} \tag{2.11}$$

2.3 Homogeneous Transformations

As the later stages of the thesis visualizes the acquired coordinates as a pointcloud in an external coordinate system, the theory behind homogeneous transformation is to be covered in this section.

Suppose that two frames are defined, namely frame a and frame b as shown in Figure 2.1. Moreover, assume that $\mathbf{R}_b^a \in \mathbb{R}^{3x3}$ is the rotation-matrix that denotes the rotation from frame a to frame b. In addition, the displacement between the two origins can be denoted as $\mathbf{r}_{ab}^a \in \mathbb{R}^3$. The notation of this vector can be read as the distance from frame a to frame b, expressed in frame a [3]. The homogeneous transformation from frame a to frame b can then be defined as

$$\mathbf{T}_b^a = \begin{bmatrix} \mathbf{R}_b^a & \mathbf{r}_{ab}^a \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{2.12}$$

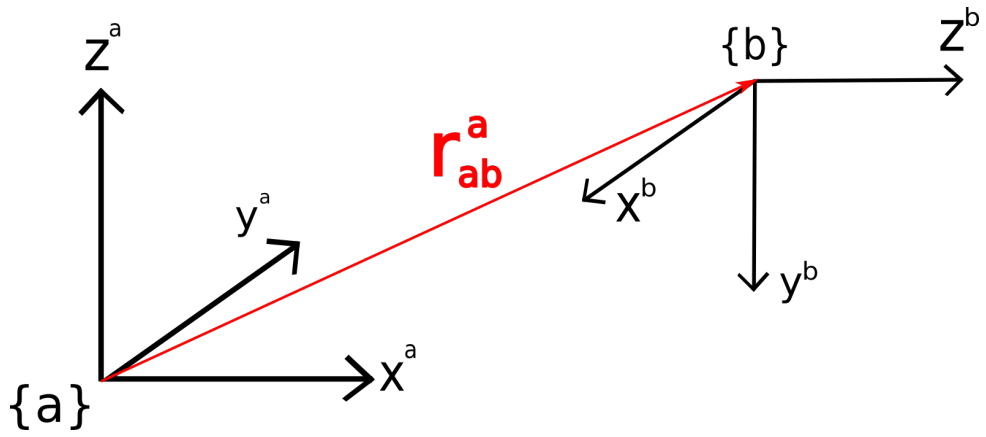


Figure 2.1: Frames a and b

Next, suppose there is a point defined in frame b, given as

$$\mathbf{p}^b = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{2.13}$$

If it is required to express this point in frame a, the homogeneous transformation matrix can be directly used to achieve this. The point expressed in frame a will be given by multiplying the homogeneous transformation with the homogeneous coordinate of the point. This is achieved by expanding the coordinate with a fourth dimension – where the value is set to one, as shown below.

$$\tilde{\mathbf{p}}^a = \mathbf{T}_b^a \tilde{\mathbf{p}}^b \quad \text{where} \quad \tilde{\mathbf{p}}^b = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.14}$$

However, if the point $\mathbf{p}$ is given with respect to frame a as opposed to frame b - the homogeneous transformation $\mathbf{T}_a^b$ has to be computed.

The inverse of the homogeneous transformation $\mathbf{T}_b^a$ can be stated as follows[3],

$$(\mathbf{T}_b^a)^{-1} = \begin{bmatrix} (\mathbf{R}_b^a)^T & -(\mathbf{R}_b^a)^T \mathbf{r}_{ab}^a \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_a^b & -\mathbf{R}_a^b \mathbf{r}_{ab}^a \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_a^b & -\mathbf{r}_{ab}^b \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{2.15}$$

Since the relationship $-\mathbf{r}_{ab}^{b} = \mathbf{r}_{ba}^{b}$ holds, (2.15) can be written as

$$(\mathbf{T}_{b}^{a})^{-1} = \begin{bmatrix} \mathbf{R}_{a}^{b} & \mathbf{r}_{ba}^{b} \\ \mathbf{0}^{T} & 1 \end{bmatrix} = \mathbf{T}_{a}^{b} \tag{2.16}$$

As demonstrated, calculating inverse of the homogeneous transformation from frame a to frame b, yields the transformation from frame b to frame a. The point $\mathbf{p}^{b}$, can then be expressed in frame a in same fashion as in (2.14), namely

$$\mathbf{p}^{b} = \mathbf{T}_{a}^{b}\mathbf{p}^{a} \tag{2.17}$$

Lastly, another frame is added to the system – frame c. Suppose frame c can be related through the following transformation.

$$\mathbf{T}_{c}^{b} = \begin{bmatrix} \mathbf{R}_{c}^{b} & \mathbf{r}_{bc}^{b} \\ \mathbf{0}^{T} & 1 \end{bmatrix} \tag{2.18}$$

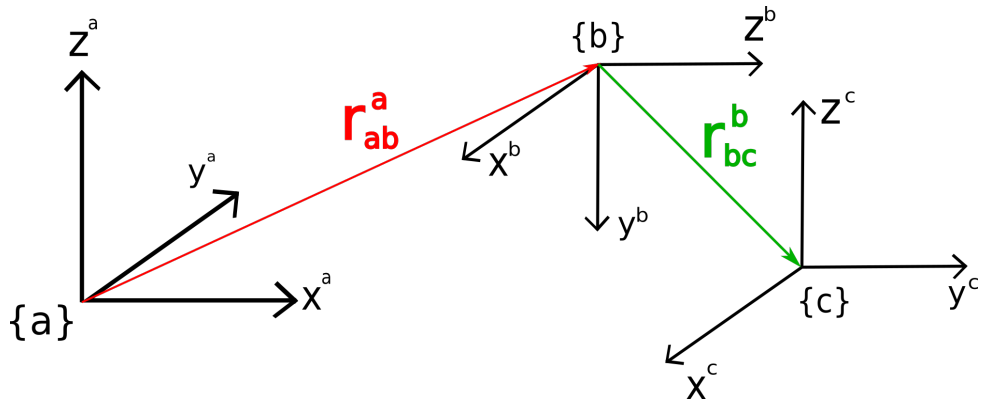


Figure 2.2: Frames a, b and c

Moreover, the homogeneous transformation from frame a to frame c can now be expressed as

$$\mathbf{T}_{c}^{a} = \mathbf{T}_{b}^{a}\mathbf{T}_{c}^{b} \tag{2.19}$$

As a result, points expressed in any of the given frames, can be transferred into the others using the method shown.

2.4 From pixel-coordinates to homogeneous coordinates

In order to use the image of a camera to identify the spatial coordinates of a point on the drill-pipe, transformations relating the pixels to metric homogeneous coordinates need to be derived. The information in this section is collected from a previous report by the author with a similar subject [5].

Moreover, the homogeneous transformation between the camera frame and the object-/world-frame is given by the following transformation

$$\mathbf{T}_{o}^{c} = \begin{bmatrix} \mathbf{R}_{\mathbf{o}}^{\mathbf{c}} & \mathbf{t}_{\mathbf{co}}^{\mathbf{c}} \\ \mathbf{0}^{T} & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{T} \\ \mathbf{0}^{T} & 1 \end{bmatrix} \tag{2.20}$$

The world coordinate system can be set arbitrarily, which means the transformation is usually known. Furthermore, it is a common convention that the z-axis of the camera frame is in the direction of where the camera-lens is pointing [6]. First, assume a point p in the camera-frame being $p = (X, Y, Z)^T$. Using a pinhole model, all rays are assumed to go through the optical center of the camera, which translates to the origin of the camera-frame [6]. By evaluating the Figure 2.3, the image coordinate $x = (x, y)^T$ can be related to the point p by considering the the focal length (f) which is the distance from the cameras origin to the image plane.

$$x = f\frac{X}{Z} \text{ and } y = f\frac{Y}{Z} \tag{2.21}$$

In the equations above, inversion of the image is cancelled by projecting the image to a plane in at $z = f$ according to Figure 2.3.

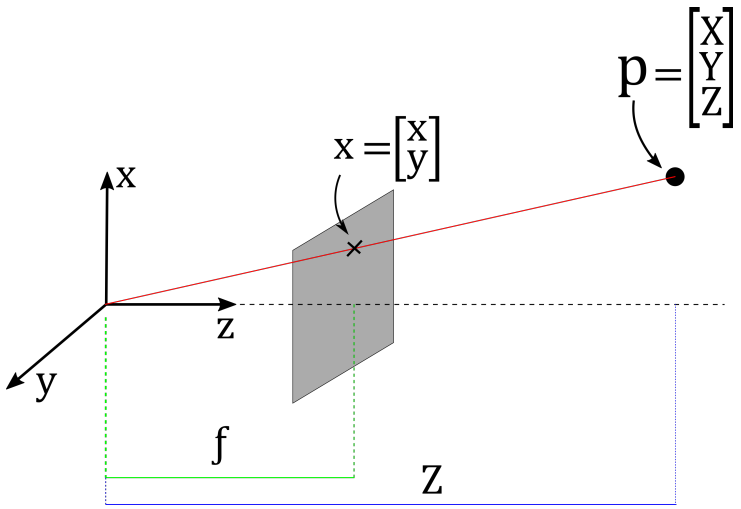


Figure 2.3: Point in the camera frame

Then, equation (2.21) is converted into homogeneous form by expanding its dimension and forming the following system.

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \frac{1}{Z}\begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \frac{1}{Z}\begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2.22}$$

Since the end goal is to actually find information about the geometry of a drill-pipe, it is not possible to use known geometry in order to derive the depth Z using the image coordinates. This will be set to an arbitrary positive constant λ. Combining equation (2.20) with (2.22), yields

$$\lambda\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} \mathbf{R} & \mathbf{T} \\ \mathbf{0}^T & 1 \end{bmatrix}\begin{bmatrix} X_o \\ Y_o \\ Z_o \\ 1 \end{bmatrix} \tag{2.23}$$

Furthermore, since images is captured in terms of pixels another transformation needs to be implemented [7]. The relationship between normalized image coordinates and pixel coordinates is given by the following transformation:

$$\tilde{\mathbf{p}} = \mathbf{K}\tilde{\mathbf{x}}$$

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\rho_w} & S_\theta & O_x \\ 0 & \frac{1}{\rho_h} & O_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2.24}$$

The two factors on the diagonal corresponds to a scale factor in the x and y direction, where ρ_w and ρ_h is the width and height of the pixels, respectively. For perfectly square pixels, these two factors will be identical. The parameter S_θ is the skew of the pixel array, which for most modern cameras can be assumed to be close to 0 [8]. The last two parameters, O_x and O_y are the coordinates of the principal point – which ideally is located at the centre of the image. These are added in order to align the center of the image to the camera's axis of view (z-axis) since the pixel coordinate $(0, 0)$ usually is set at the top left of an image.

Combining (2.23) and (2.24) yields the final transformation between pixels-coordinates and the world frame:

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\rho_w} & S_\theta & O_x \\ 0 & \frac{1}{\rho_h} & O_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{T} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} X_o \\ Y_o \\ Z_o \\ 1 \end{bmatrix}$$

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{f}{\rho_w} & 0 & O_x \\ 0 & \frac{f}{\rho_h} & O_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{T} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} X_o \\ Y_o \\ Z_o \\ 1 \end{bmatrix} \tag{2.25}$$

The first matrix in the linear equation system 2.25 is called the intrinsic parameter matrix or calibration matrix which is initially unknown [7]. In order to identify these parameters, a camera calibration needs to be carried out, which will be described in the section 2.7.

2.5 Robot welding

In the request of having tasks completed in a more efficient and more consisted fashion, more and more industries are implementing the use of robots in their production line. Although the robot can not replace humans in all aspects of the production line, repetitive or mundane tasks such as assembling or quality control are tasks well suited for a robot. Furthermore, as the technology keeps improving, the horizon of tasks performable by a robot keeps expanding. Since the first welding robot was introduced in the 1960, the technology has seen large improvements in terms of quality and speed [9]. The advantages of using a robot to perform a weld are not necessarily only connected to speed and quality, but to safety as well. In some cases, welding fumes may cause harm to the workers – which is not desired. As a robot is unaffected by the fumes, this danger is eliminated. However, due to potential malfunctioning of the robot, precautions must be made to prevent injury to the workers. Studies show that workers have a tendency to underestimate the reach of a robot, which can lead to accidents [10].

Furthermore, in order for a robot to perform a weld, it must be given the correct input. Some robots performs the weld by a preprogrammed path [11], and is preferred in situations where the relationship between the robot and object does not vary to a large extent. However, in situations where the conditions can not be regarded as constant, additional data has to be provided to the robot. In this thesis, the additional data will be provided using laser triangulation, which is the topic of the next section.

Moreover, the required accuracy of the visual data depends on the type of welding. For spot-welding an acceptable error is in the range of 1 mm. However, if the robot is to perform arc-welding, the accuracy should be less than 0.5 mm [12]. As the welding relevant in this thesis is arc-welding, the distance of 0.5 mm will be used as a benchmark in which the systems accuracy will be compared to.

2.6 Laser triangulation

Since the use of robots in various production lines has increased in the last years [13], so has the request for supplying the robots with visual inputs. Providing the robots with visual inputs, introduces a potential of automation which is not achievable with preprogramming methods.

One of the most frequently used method for providing a robot with a visual input is laser triangulation. This method relies mainly on two components, an emitter and a sensor. As the emitter projects light upon the surface of an object - the sensor registers the distorted light pattern caused by objects curvature. The pattern of the emitted light varies by its application. If the purpose of the scan is to make a 3D-representation of a static object, a pattern which consist of several laserlines are commonly used as this provide more spatial points for each pose of the object. However, if the object can be rotated or translated through the projection of the laser, the same amount of reading can be achieved with a single laserline [14], assuming the rotation or translation can be measured. As for the system of this thesis, the object is rotated while the laserplane is intersecting its surface. For this reason, laser triangulation with a single laserline is utilized in the experiments of this thesis.

Assuming the pixels which corresponds to the laser on the image can be identified, a sense of direction to the given point in space can be calculated as shown in Figure 2.4. However, as this direction is not sufficient to determine the distance to the point, triangulation needs to be implemented. If the position and orientation of the laserplane is known, the point of intersection between the line spanned by the pixel and the plane of the laser can be calculated. This intersection yields the three dimensional coordinate located at the surface of the object.

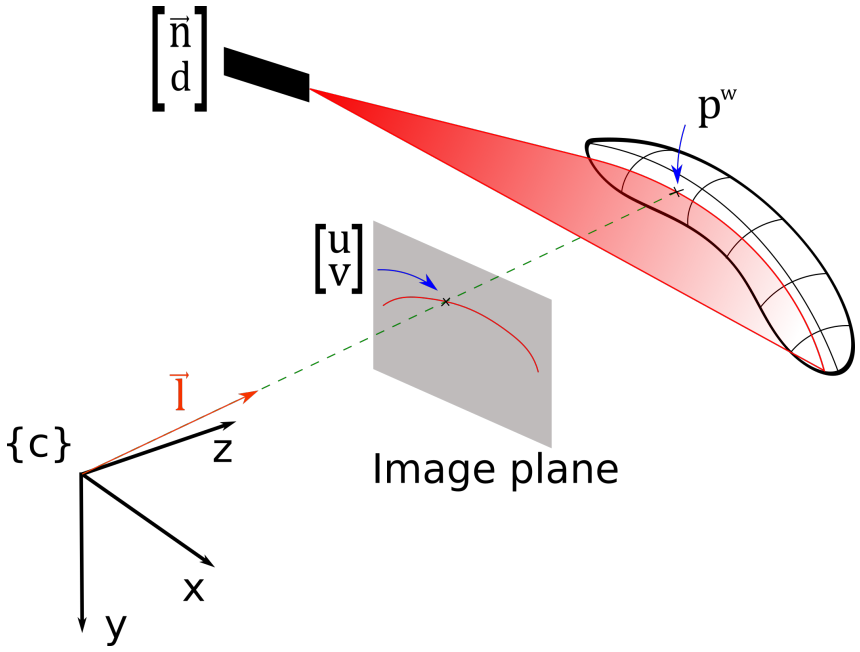


Figure 2.4: Laser triangulation

However, since the location and orientation of the laser with respect to the camera is not necessarily trivial, and needs to be computed with care in order to achieve accurate results. For this reason, a significant portion of this thesis is to estimate the relationship between the laserplane and the camera with respect to position and orientation, also known as a laserplane calibration.

Moreover, the triangulation process is in this regard only possible for pixels which represent the laser on the image plane. Consequently, the pixels related to the laserline must be identified in terms of pixel-coordinates, which is the purpose of the camera described in a later section.

2.7 Camera calibration

Due to imperfections in the camera, more specifically in the lens – pictures are bound to be distorted to some extent [6]. When using a camera as an input for a system, these distortions can lead to errors in the calculations of trajectories. This section describes how these imperfections can be identified by using images of checkerboards. Some of the information in this section is collected from the authors specialization report [5], where OpenCV's toolbox was used for the camera calibration.

Moreover, there are mainly two types of distortion – namely radial and tangential distortion. The radial distortion is a result of the curvature of the lens, and tangential distortion is the result of the lens not being perfectly parallel to the imaging plane [6]. According to the book *Computer Vision* [15], radial distortion can be modelled as a low-order polynomial:

$$\hat{x} = x(1 + \kappa_1 r^2 + \kappa_2 r^4) \tag{2.26}$$

$$\hat{y} = y(1 + \kappa_1 r^2 + \kappa_2 r^4) \tag{2.27}$$

Where κ_1 and κ_2 are called the radial distortion parameters - and r^2 is equal to $x^2 + y^2$.

Furthermore, according to *Robotics Vision and Control* [6] the tangential distortion can be represented using the following polynomial, where p_1 and p_2 are the tangential distortion coefficients,

$$\hat{x} = 2p_1xy + p_2(r^2 + 2x^2) \tag{2.28}$$

$$\hat{y} = p_1(r^2 + 2y^2) + 2p_2xy \tag{2.29}$$

Combining these two equations results in the total distortion.

$$\hat{x} = x(1 + \kappa_1 r^2 + \kappa_2 r^4) + [2p_1xy + p_2(r^2 + 2x^2)] \tag{2.30}$$

$$\hat{y} = y(1 + \kappa_1 r^2 + \kappa_2 r^4) + [p_1(r^2 + 2y^2) + 2p_2xy] \tag{2.31}$$

The unknown distortion coefficients of the equations above can be represented as the vector,

$$\mathbf{D}_c = \begin{bmatrix} \kappa_1 & \kappa_2 & p_1 & p_2 \end{bmatrix} \tag{2.32}$$

By carrying out a camera calibration, these distortions-coefficients can be approximated in addition to the intrinsic parameters of the camera which was mentioned in section 2.4. This process is usually only required once, unless the camera experiences trauma that influences the lens, and hence changes the intrinsic parameters. The software used in this thesis is the calibration app "Camera Calibrator" of Matlab, which is included in the Computer Vision package [16]. The calibration app offers a simple user-interface, and includes visuals which may ease the process. In addition, the software allows expansion of the polynomial of the radial distortion with a third element. However, as this is usually only required for lenses with high distortion (e.g fisheye lenses) [17] – the third coefficient will not be utilized in this thesis.

Moreover, in order to calibrate a camera, several pictures needs to be taken of a standardized pattern. The algorithm uses the known geometry of the pattern, and estimate the parameters based on the given images. This method is otherwise known as the *plumb line method* [15]. For the calibrations in this thesis, different sizes of a 7x9 checkerboard is used, as shown in Figure 2.5. Note that the pattern size is not determined by the number of squares, but the number of points centered in each 2x2.

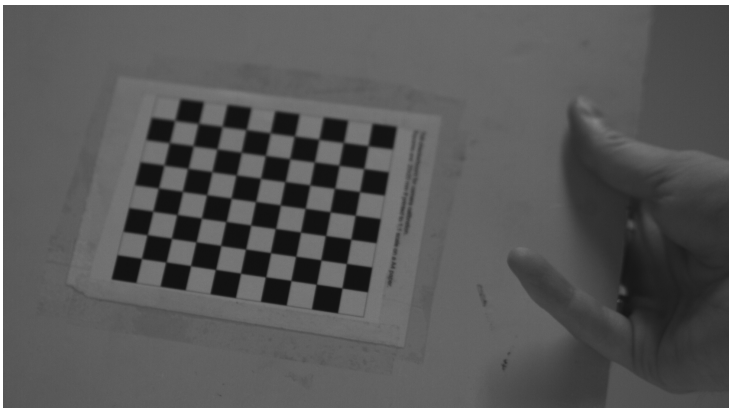


Figure 2.5: 10mm 7x9 checkerboard pattern

According to the documentation of the software, at least 10 - 20 different images should be used in order to achieve a well posed system [17] with better estimation accuracy. The checkerboards in the images should differ both in orientation and distance from the camera, and should contain all of the interest-points of the pattern. Additionally, if the checkerboard is printed on paper as it was in this thesis, it is important that the paper is attached to a straight and sturdy surface. If the pattern is curved due to an uneven surface, the curvature will be interpreted as lens-curvature and yield inaccurate results. The software also requires a measurement of the squares on the checkerboard. This measurement does not effect the calculation of the intrinsic values (i.e K-matrix and distortions), but if the relative position of the checkerboard with respect to the camera frame is needed, this dimension must be measured precisely. The choice of the checkerboard metric size depends mainly on the focus of the camera (i.e the field of view). For a larger FOV it is favorable to use a larger checkerboard, such that the points are more easily detected by the camera.

2.7.1 Verification of a calibration

In order to identify if the calibration is successful, there are built-in visuals in the software that can be analyzed to determine the quality of the calibration.

The first thing that can be verified is that the positions of the checkerboards are somewhat in line with the expectations. As shown in Figure 2.6, the software offers a view that relates the checkerboards position and orientation with respect to the camera frame. By inspecting the figure, it is possible to discover obvious errors which might occur during the process. If one of the checkerboards were to be located unrealistically far away from the others, or even behind the camera – the checkerboard can be removed before initiating the calibration process once more. Note that the dimensions along the axis are only correctly displayed if the square-size are correctly measured.

Additionally, the visualization can also be used to establish if the position and orientation between the checkerboards differ enough to provide good input data for the calibration process.

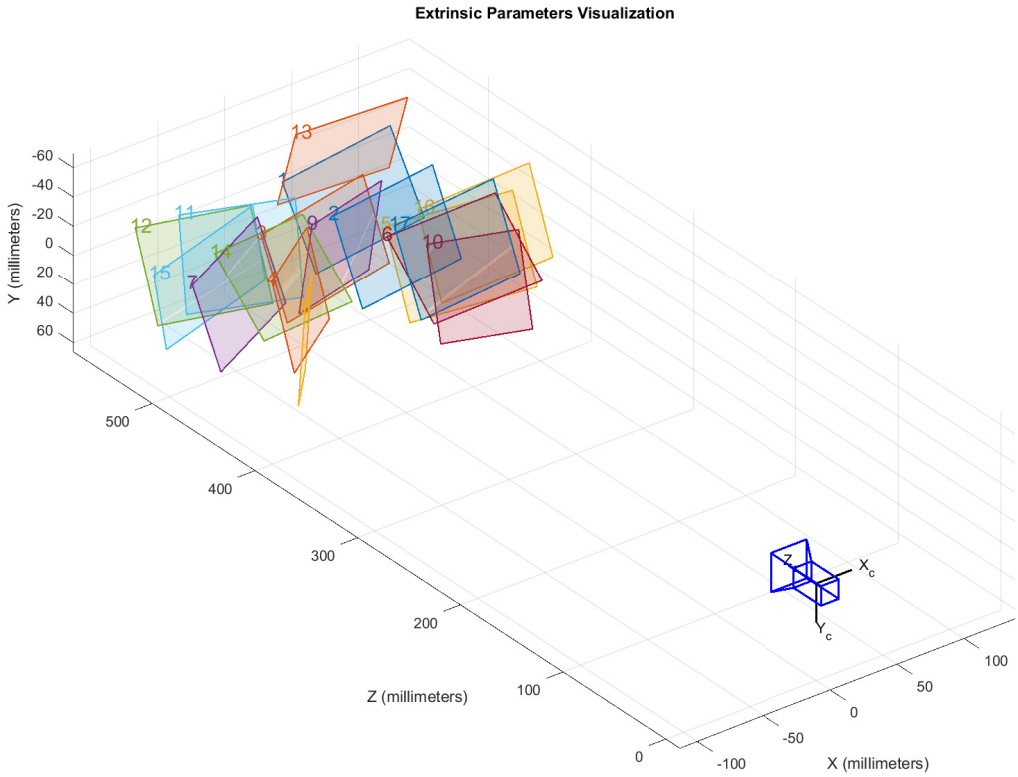


Figure 2.6: Checkerboards in camera frame

Furthermore, in order to determine a quantitative value for the calibrations accuracy the reprojection errors can be analyzed. By inspection of Figure 2.7, the reprojection error is the distance in pixels from the detected point, and the point reprojected by the software based on the estimated intrinsic and extrinsic parameters [8].

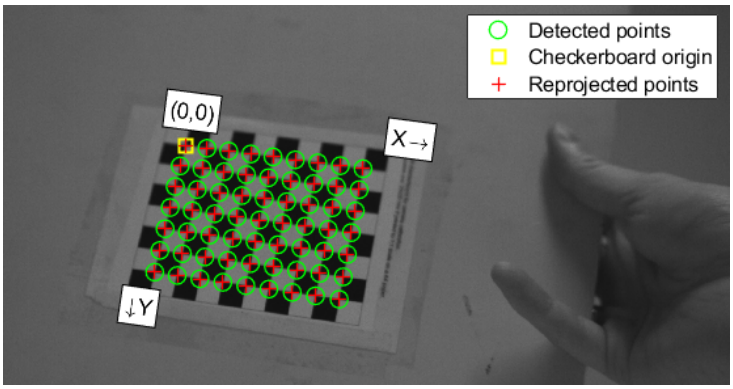


Figure 2.7: Identified points on 10mm checkerboard

Figure 2.8 shows the average of the reprojection error corresponding to each image. If there are images that contribute to a large mean reprojection error, they can be identified by the height of their corresponding column before they are removed from the calibration. Generally, a mean reprojection error below 1 pixel is accepted [18], but should be minimized when possible by removing the images with high reprojection error from the calibration.

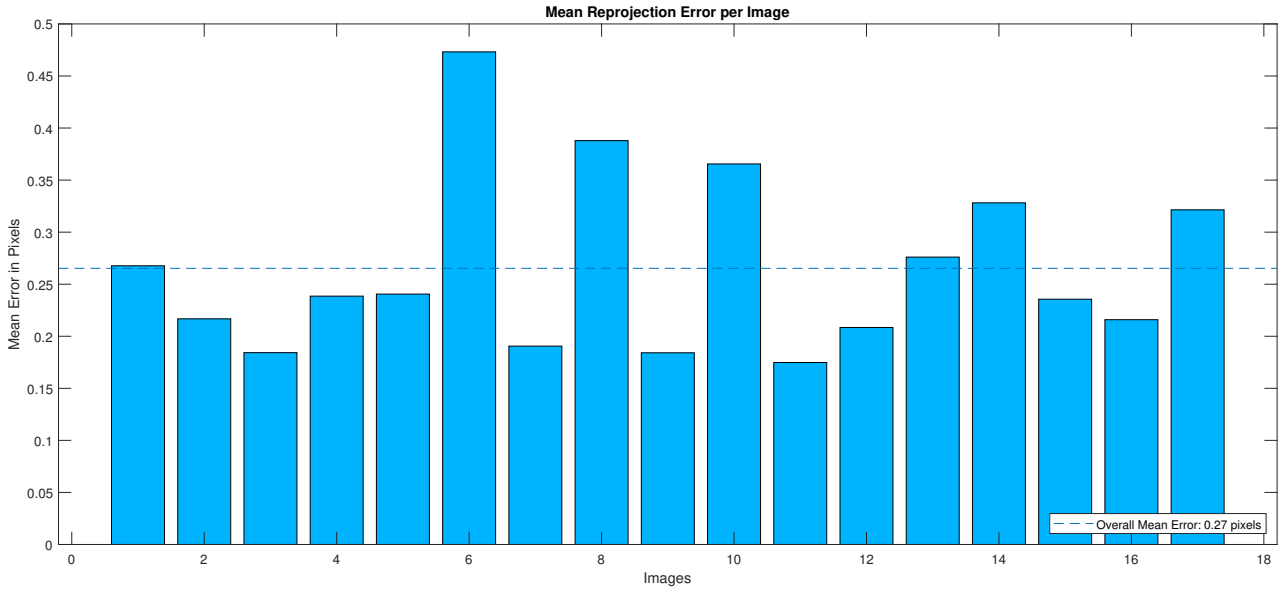


Figure 2.8: Reprojection error

2.7.2 Data from the calibration

After the process described above is completed, all the identified parameters are accessible through the workspace of Matlab.
The intrinsic parameter matrix is given by the software to be,

$$\mathbf{K} = \begin{bmatrix} \frac{f}{\rho_w} & 0 & 0 \\ 0 & \frac{f}{\rho_h} & 0 \\ O_x & O_y & 1 \end{bmatrix} \tag{2.33}$$

By comparing with the intrinsic matrix from (2.25), it can be seen that this matrix is the transpose of the matrix earlier stated. This is due to the software using another formulation than the one derived in this thesis. More specifically, the world coordinates are stated as a row-vector instead of the column vector in 2.25. Due to this, the matrix is transposed before extracted to the corresponding text-file. Moreover, the software does not provide an absolute value for neither the focal length f of the pixel-dimensions ρ_w and ρ_h, as only the ratio between them is estimated. However, as most camera-manufacturers state the dimensions of the pixels, the focal length can be explicitly calculated if needed.

Furthermore, the estimated distortion coefficients are stated separately as two 2x1-vectors containing the coefficients stated in the distortion-vector 2.32. These are later combined in to one vector, as it is required by the function which undistorts the pixels.

$$\mathbf{r}_{dist} = \begin{bmatrix} \kappa_1 & \kappa_2 \end{bmatrix}$$

$$\mathbf{t}_{dist} = \begin{bmatrix} p_1 & p_2 \end{bmatrix}$$

In addition, extrinsic parameters related to the orientation and position of the checkerboards is given by the software. This is the data which is used by the software in order to illustrate the Figure 2.6 shown previously. These are not key elements of a standard camera calibration, but may in some instances prove to be useful. In this thesis both the translational vectors and rotation matrices are directly used in order to generate a pointcloud of different points on the laserline as a part of the laserplane calibration. More specifically, the data was used to express the checkerboards as planes by using point-normal notation, which is the subject of section 2.8.

2.7.2.1 Translational vectors

The given translational vector for each checkerboard yields the distance from the cameras origin to the checkerboards origin, where the latter is shown in Figure 2.7 as a yellow square in the top-left corner. The unit of the elements in this vector depends on which unit the square size of the checkerboard is given in.

$$\mathbf{t}_{checker}^{cam} = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \tag{2.34}$$

2.7.2.2 Rotation matrices

In addition to the translation of the checkerboards, the software also calculates the relative orientation between the checkerboard and the camera frame. Due to notation used by Matlab, the rotation is expressed as the rotation from the calibration object (i.e the frame of checkerboard) to the frame of the camera. The orientational relationship between the frame of the camera and the checkerboard is stated as the following rotation matrix.

$$\mathbf{R}_{cam}^{checker} = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \tag{2.35}$$

However, as stated in section 2.1.1 the rotation matrix is a member of $SO(3)$. As the relationship $\mathbf{R}^T\mathbf{R} = I$ holds, the checkerboards orientation expressed with respect to the camera frame can be simply retrieved by transposing the matrix.

$$\mathbf{R}_{checker}^{cam} = (\mathbf{R}_{cam}^{checker})^T = \begin{bmatrix} R_{11} & R_{21} & R_{31} \\ R_{12} & R_{22} & R_{32} \\ R_{13} & R_{23} & R_{33} \end{bmatrix} \tag{2.36}$$

If the matrix is compared to the image shown in Figure 2.7, the direction of the three axes of the checkerboard can be identified by inspecting the column-vectors of the matrix.

$$\mathbf{R}_{checker}^{cam} = \begin{bmatrix} \overline{\mathbf{x}} & \overline{\mathbf{y}} & \overline{\mathbf{z}} \end{bmatrix} \tag{2.37}$$

Additionally, by applying the right hand rule to the x- and y-axis of the checkerboard in Figure 2.7, the z-axis is defined to be in the direction opposite of the camera.

2.7.2.3 Rotation errors

The software also provides the error of the checkerboards apparent orientation. This parameter is given as a vector, which represents the uncertainty of the rotation stated on axis-angle form. Moreover, the uncertainty can be quantified by the length of the error-vector. The greater the length of the vector is, the more uncertainty is associated with the given orientation.

2.7.2.4 Translational errors

Similar to the estimated orientations of the checkerboards, the translational vectors also has uncertainties. This error is given in the same units as the translational vector, and contains three different values corresponding to the uncertainty in the x, y and z-direction.

2.7.3 Python accessibility

As it was necessary to access the identified parameters through Python at later stages of the thesis, the parameters had to be extracted from Matlab. After the images has been processed by the calibration software, the identified parameters were exported to Matlab's workspace as variables. The variables were then saved individually as text-files, which was a file type compatible with Python. A function in Python was then written in collaboration with Trym D. Hauglund in order to access these files from an arbitrary Python-script. The function was called *loadCaliParam* and enabled a clean and efficient way of accessing the calibration parameters from external scripts. The function is shown in appendix 1, and is able to read and supply the values from the text-file of the parameters.

2.8 Plane representation

As a major part of this thesis is to represent the plane spanned by the laser projector in the frame of the camera, some of the theory behind planes and their different representations are to be presented in this section.

The standard notation of representing a plane in $\mathbb{R}^3$ is given by

$$Ax + By + Cz + D = 0 \tag{2.38}$$

The coefficients A,B,C describes the slope of the plane in the directions x,y and z, respectively. The constant D describes the Euclidean distance from the origin of the coordinate system to the plane.

Moreover, in some cases it is more applicable to use point-normal notation instead, as shown in the next section. The point-normal representation consist of two elements, namely the normal of the plane in addition to a point present on the plane. The normal of the plane can be computed by the coefficients of the variables in (2.38) as follows.

$$\mathbf{n} = \begin{bmatrix} A \\ B \\ C \end{bmatrix} \tag{2.39}$$

Furthermore, the point which is used in this representation can be an arbitrary point on the plane. Depending on the representation, the simplest point to calculate is the intersection of one of the axes. For the intersection of the z-axis, both x and y is set to zero. The remainder of (2.38) is $Cz + D = 0$, where the point on the plane can be expressed with,

$$d = \begin{bmatrix} 0 \\ 0 \\ -\dfrac{D}{C} \end{bmatrix} \tag{2.40}$$

For simplicity, a function was written in Python which converted from one representation to the other. The function detects which representation it is given, and converts to the other. The function was called *planeify* and is found in appendix 14.

2.9 Line-plane intersection

As the theory of line-plane intersection was to be used both in the laser triangulation and the laserplane calibration, the theory behind it will be briefly explained in this section.

As shown in the previous section, a plane can be represented as a point in addition to a vector normal to the plane like such

$$(p - p_0) \cdot \mathbf{n} = 0 \tag{2.41}$$

Where p is an arbitrary point on the plane, p_0 is a known point on a the plane and $\mathbf{n}$ is the vector normal to the plane. Furthermore, if a line has a known point and direction, it can be written in the following form

$$p = d\mathbf{l} + l_0 \tag{2.42}$$

Where p is an arbitrary point on the line, $\mathbf{l}$ is the direction of the line and l_0 is a point known to be on the line. The real numbered scalar d will in this context be the distance from the point on the line to the plane - given that $\mathbf{l}$ has a length equal to 1 (i.e normalized).

Moreover, if (2.42) is inserted into (2.41), the following equation yields

$$(d\mathbf{l} + l_0 - p_0) \cdot \mathbf{n} = 0 \tag{2.43}$$

Solving (2.43) for the scalar d, the equation becomes

$$d = \frac{(p_0 - l_0) \cdot \mathbf{n}}{\mathbf{l} \cdot \mathbf{n}} \tag{2.44}$$

Lastly, as equation (2.44) gives a value for d, (2.42) can be used to determine the intersection point in spatial coordinates.

2.10 Plane fitting

During the process of the laserplane calibration, the plane spanned by the laser will be represented in the shape of a pointcloud. However, as the parameters of the laserplane needs to be estimated, this pointcloud needs to be further processed in order to be used in the triangulation process. By using plane fitting methods, the parameters of the laserplane can be estimated based on the shape of the pointcloud, which has the following form.

$$\mathbf{P} = \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ & \vdots & \\ x_n & y_n & z_n \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{bmatrix} \tag{2.45}$$

2.10.1 Least squares

The first method that was used, was to estimate the plane-parameters was a least squares-method. The method involves representing the pointcloud as a over-determined linear system $A\mathbf{x} = \mathbf{b}$. In this representation, the matrix A contains the x and y coordinates, $\mathbf{x}$ is the parameters of the plane- and $\mathbf{b}$ will be the corresponding z-values.

Furthermore, the pseudo-inverse of the system-matrix is then used in order to estimate the plane in which the points lie. According to Lieven Vandenberghe [19], this method optimizes the least squares problem

$$\min \quad ||(A\mathbf{x} - \mathbf{b})||^2 \tag{2.46}$$

In other words, the plane is placed such that it minimizes the sum of errors between the estimated planes z-values and the pointcloud as illustrated in Figure 2.9.

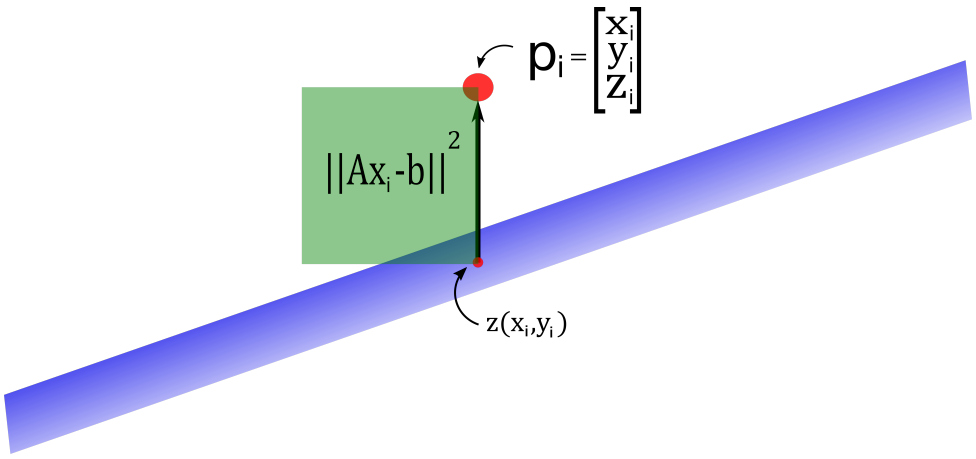


Figure 2.9: Least squares of z-value

Moreover, since the laser projector forms a plane it can be assumed it can be expressed using the standard notation, namely

$$Ax + By + Cz + D = 0 \tag{2.47}$$

This can be rewritten as

$$\tilde{A}x + \tilde{B}y + \tilde{D} = -z \tag{2.48}$$

Where,

$$\tilde{A} = \frac{A}{C} \quad , \quad \tilde{B} = \frac{B}{C} \quad , \quad \tilde{D} = \frac{D}{C}$$

Furthermore, since

$$\begin{bmatrix} x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots \\ x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} \tilde{A} \\ \tilde{B} \\ \tilde{D} \end{bmatrix} = \begin{bmatrix} -z_1 \\ \vdots \\ -z_n \end{bmatrix} \quad , \quad \mathrm{A}\mathbf{x} = \mathbf{b} \tag{2.49}$$

As the system (4.6) is over-determined, the vector $\mathbf{x}$ can not be simply derived by performing a left-sided inverse multiplication of the A^{-1}-matrix. However, by doing a left-sided multiplication of A^T, we get the following expression.

$$\mathrm{A}^T\mathrm{A}\mathbf{x} = \mathrm{A}^T\mathbf{b}$$

As the A-matrix is of the dimension nx3, and its transpose A^T is 3xn, the product of $A^T A$ will be a 3x3 matrix. Assuming full rank, the resulting matrix can be inverted – and we reach the final expression.

$$\mathbf{x} = (\mathrm{A}^T\mathrm{A})^{-1}\mathrm{A}^T\mathbf{b} = \mathrm{M}\ \mathbf{b} \tag{2.50}$$

The matrix M is called the left pseudo matrix, but is more commonly called the Moore-Penrose inverse [20]. Using this expression, the plane can be estimated by evaluating the values of $\mathbf{x}$.

2.10.2 Optimization using orthogonal distances

As the method in the previous section was a least squares solution to the difference in function value, the author wanted to derive a solution for the distances normal to the plane. Moreover, the problem of fitting a plane to the extracted coordinates, could be seen as a optimization problem where the goal is to determine a plane in which the orthogonal distance between the plane and the points is at its lowest.

Given the pointcloud (2.45), its centroid can be calculated as

$$c = \frac{1}{N}\sum_{i=1}^{N} p_i \tag{2.51}$$

The *shortest* distance between any given point and a plane is the distance in the normal direction of the plane, as shown in Figure 2.10. If it is assumed that the centroid of the plane lies on the plane that is to be estimated, the distance between an arbitrary point and the plane will be given by

$$d = (p_i - c)^T \cdot \mathbf{n} \tag{2.52}$$

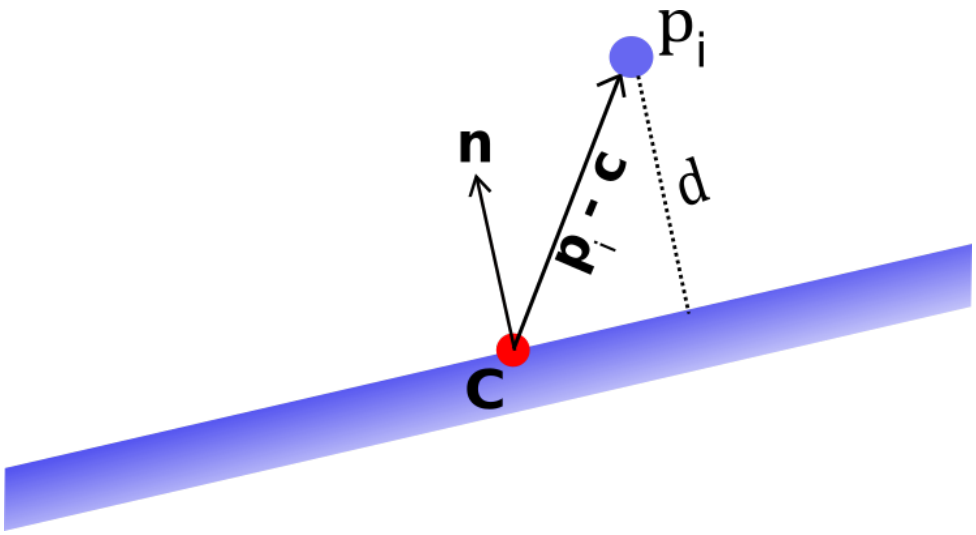


Figure 2.10: Distance in normal direction

Using (2.52), the cost function can then be defined. As the desired plane is placed such that the sum of all distances between the points and the plane is at its minimum, the cost function becomes

$$\min_{\mathbf{n}} \sum_{i=1}^{N} ((p_i - c)^T \cdot \mathbf{n})^2 \tag{2.53}$$

Naturally, the distances are squared to avoid negative values.

Moreover, without any constraints the sum of the cost function 2.53 would show as zero, and the solution would be stated as

$$\mathbf{n} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

In order to prevent the zero-vector of being the solution of the optimization problem, an equality-constraint must be defined which forces the vector of having an actual direction. This can be done by applying the following constraint to the optimization-problem, which requires the length of the normal-vector to be 1.

$$n_x^2 + n_y^2 + n_z^2 = 1 \tag{2.54}$$

In turn, the normal-vector is constrained to the surface a unit-sphere where the length is equal to one for any arbitrary orientation.

The optimization problem can then be stated as

$$\begin{aligned} \min_{\mathbf{n}} & \sum_{i=1}^{N} ((p_i - c)^T \cdot \mathbf{n})^2 \\ \text{s.t} \quad & n_x^2 + n_y^2 + n_z^2 = 1 \end{aligned}$$

3. Experimental setup

As the purpose of the thesis is to combine an angular measurement of an object with its corresponding curvature along a laserline, the experiments can be divided into three main parts.

- Acquiring an angular measurement
- Laserplane calibration with accuracy verification
- Combining the two measurements and performing a rotary scan

This chapter will describe the various equipment and software that was needed to perform the experiments of this thesis. In this regard, the camera that was used in the extraction of pixel-coordinates will be given a thorough explanation in terms of its functionality and algorithms. In addition, the configuration between the camera and laser projector will be given a description together with the formula regarding its theoretical resolution.

3.1 Camera

In order to locate the pixels related to the laserline on the image a suitable camera had to be used. The camera used in the experiments of this thesis is the C4-2040-GigE camera manufactured by Automation Technology (AT), and is shown in Figure 3.1. The camera has a resolution of 2048x1088 pixels, and is designed specifically for 3D measurement purposes. Furthermore, the camera comes pre-installed with software tailored for pixel-coordinate extractions. This section will explain how the camera operates, some of its functions, and how it is able to provide pixel-coordinates of a laser located in the image. The information in this section is mainly collected from the manual of the camera [21], and the images shown is with the courtesy of AT.

Figure 3.1: C4-2040 GigE, Source : AT

By analyzing the different columns of pixels and reading the intensity of each individual pixel, the camera is able to pin-point the location of a laserline present in its image. Naturally, this builds on the assumption that the highest intensity for each column is due to the laser and not external light-sources. If the intensity of pixel-column N is analyzed, and the center of the laserline is located at pixel number M the extracted pixel coordinate will be $[N, M]$. As shown in Figure 3.2, the intensity along the pixel-column marked with red is analyzed. The software used in the figure is AT's CXExplorer, which enables interactive analysis of the reading. This software was used for illustrative purposes and is not used explicitly in the experiments in this thesis.

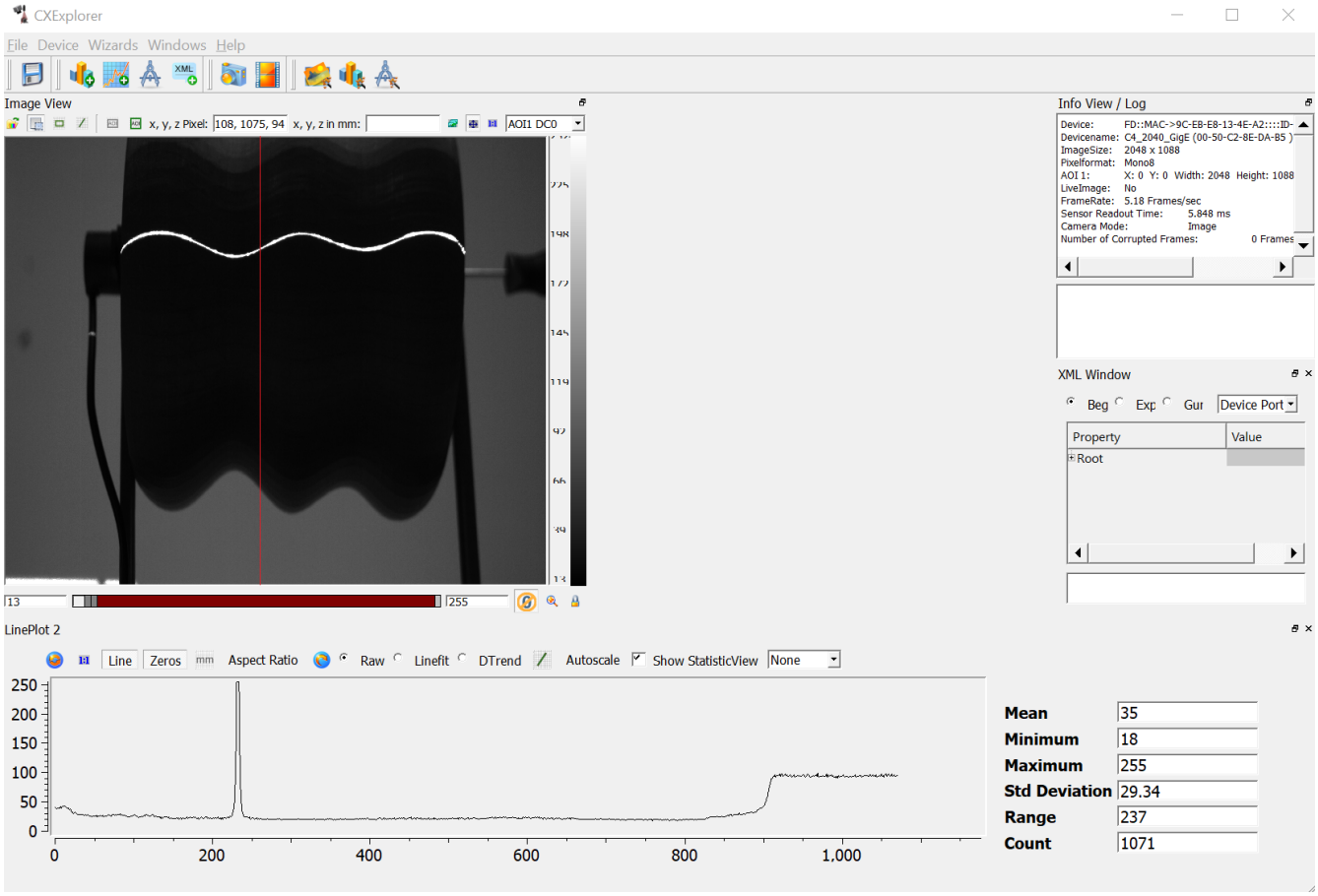


Figure 3.2: Screenshot from CXExplorer

Moreover, as seen from Figure 3.2 the intensities along the pixel-column are plotted below the image in *LinePlot 2*. As the image captured by the camera is an 8-bit gray scaled image, the intensity value ranges from 0 (black) to 255 (white). The peak of intensity in the plot corresponds to the laser, and can be approximately read as pixel number 232.

Naturally, as it is not desirable to manually read out the pixel-coordinates of the laser – the camera is equipped with algorithms which automatically outputs the pixel-coordinates. These algorithms are explained in later sections.

3.1.1 Finite Impulse Response Filter

The camera had also an integrated FIR filter in the form of Savitzky-Golay filter with up to 9 coefficients [22]. The filter is used to smooth out the intensity profile, which can improve the accuracy of the algorithms explained in the following sections. By using the following formula each intensity-reading is recalculated based on the adjacent values of intensities [23].

$$Y_j = \sum_{n=-M}^{M} b_i y_{j+i} \tag{3.1}$$

The neighbourhood of values used in the filter depends on the value M, whereas $2M + 1$ is the number of points evaluated in each iteration. For $M = 4$, each point is recalculated based on itself, the four previous and the four next values in the data set. Each of these values are multiplied with its corresponding coefficient.
As a result, the point is recalculated based in the trend of the neighbouring intensities, such that sudden changes of values is suppressed. The intended effect of the filter is illustrated in Figure 3.3.

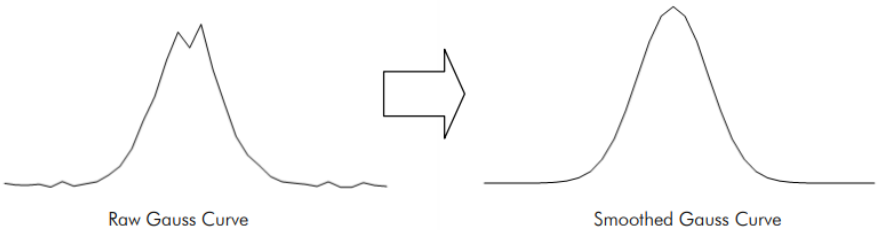


Figure 3.3: Smoothing the curve

Furthermore, the coefficients of the filter can be set in such a manner that instead of smoothing the curve, it approximates the slope of the curve. These coefficients are used in the FIR-Peak algorithm, which is one of the algorithms that will be briefly explained in the next sections.

In this regard, AT was contacted in order to gain the exact values for the different filters. AT replied saying that these were confidential, and a core part of their know-how. However, they where able to provide the following image which shows a crude representation of how the coefficients are set.

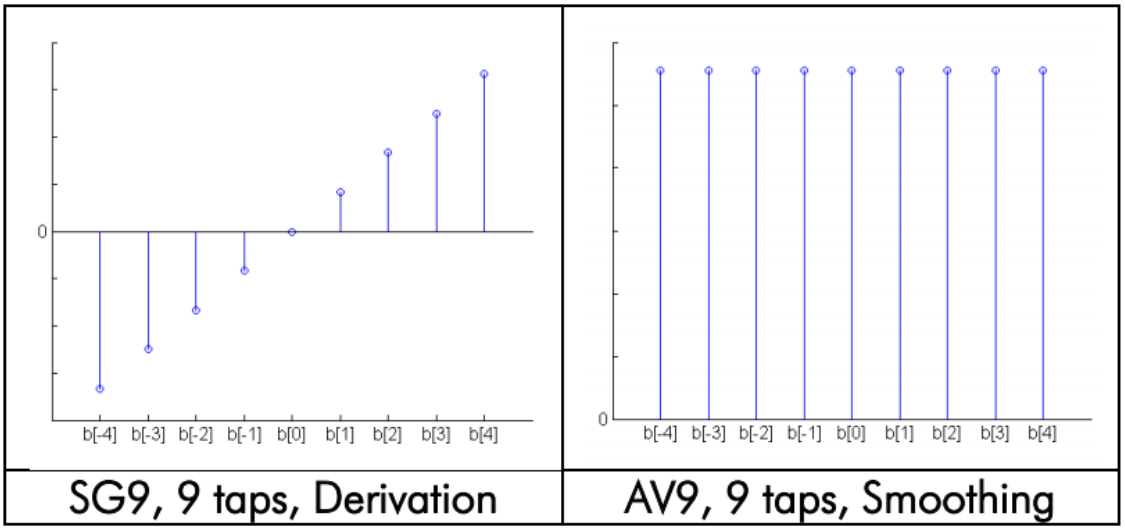


Figure 3.4: FIR coefficients

There are four different main algorithms which calculates the pixel-coordinates based on the array of intensity-values shown in Figure 3.2.

3.1.2 Threshold (TRSH)

By using the TRSH algorithm, the two points P_L and P_R are registered, which are shown in Figure 3.5. The intensity threshold AOI_TRSH can be set as desired, and the algorithm will ignore intensity values that are lower than the set value. The extracted pixel-coordinate for the given pixel-column is then calculated by

$$P_{TRSH} = \frac{P_L + P_R}{2} \tag{3.2}$$

As seen from (3.2), the algorithm assumes that the center of the laser is directly in the middle of the pixel P_L and P_R. Furthermore, as both P_L and P_R are integers, resolution of this algorithm will be half a pixel.

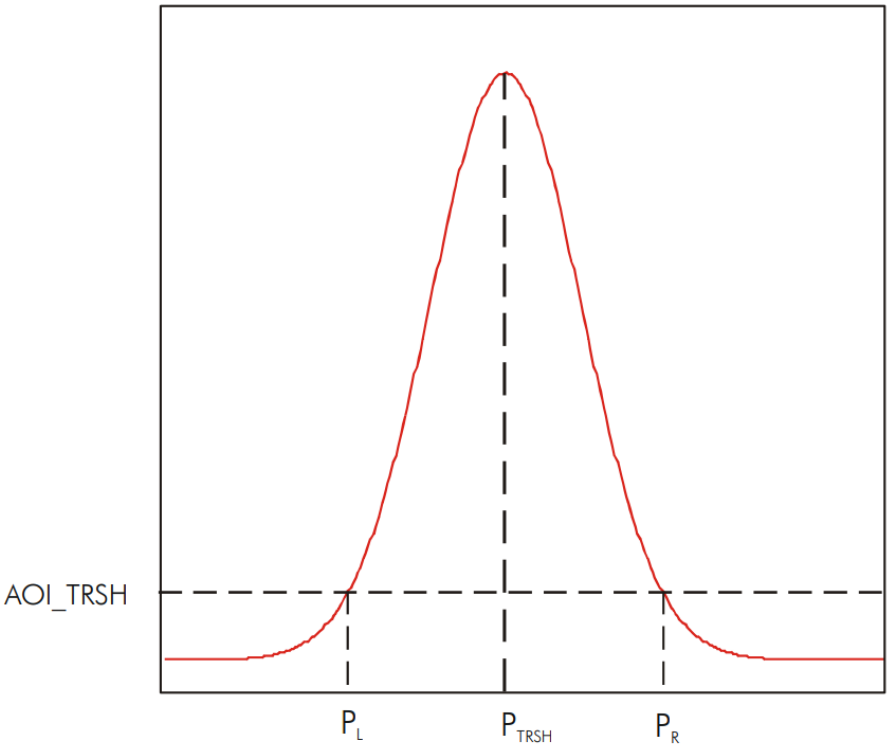


Figure 3.5: TRSH-algorithm

3.1.3 Maximum intensity (MAX)

Next, there is the MAX algorithm which select the pixel corresponding to the highest intensity value in the pixel-column as demonstrated in Figure 3.6. This is the simplest of the algorithms as this does not require any further computation. Furthermore, as the pixel-coordinate is directly collected from the intensity array, the resolution of this algorithm is one pixel.

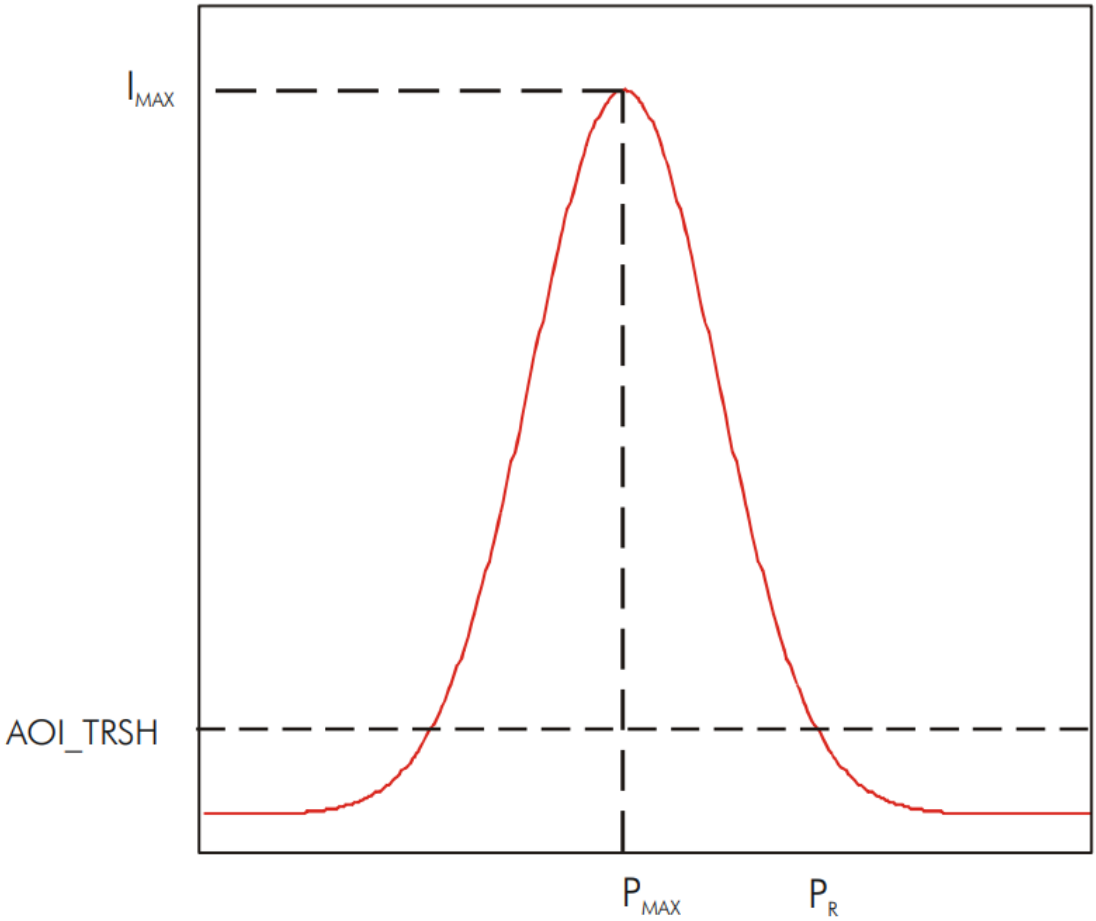


Figure 3.6: MAX-algorithm

However, if the optical sensors of the camera becomes saturated there is a risk for several maximums as illustrated in Figure 3.7. This poses a challenge as the algorithm automatically reads the first point of saturation as P_{MAX}, which often will result in an inaccurate reading of the lasers position. However, the camera has software to handle problems regarding saturation – which will be briefly explained in a later section.

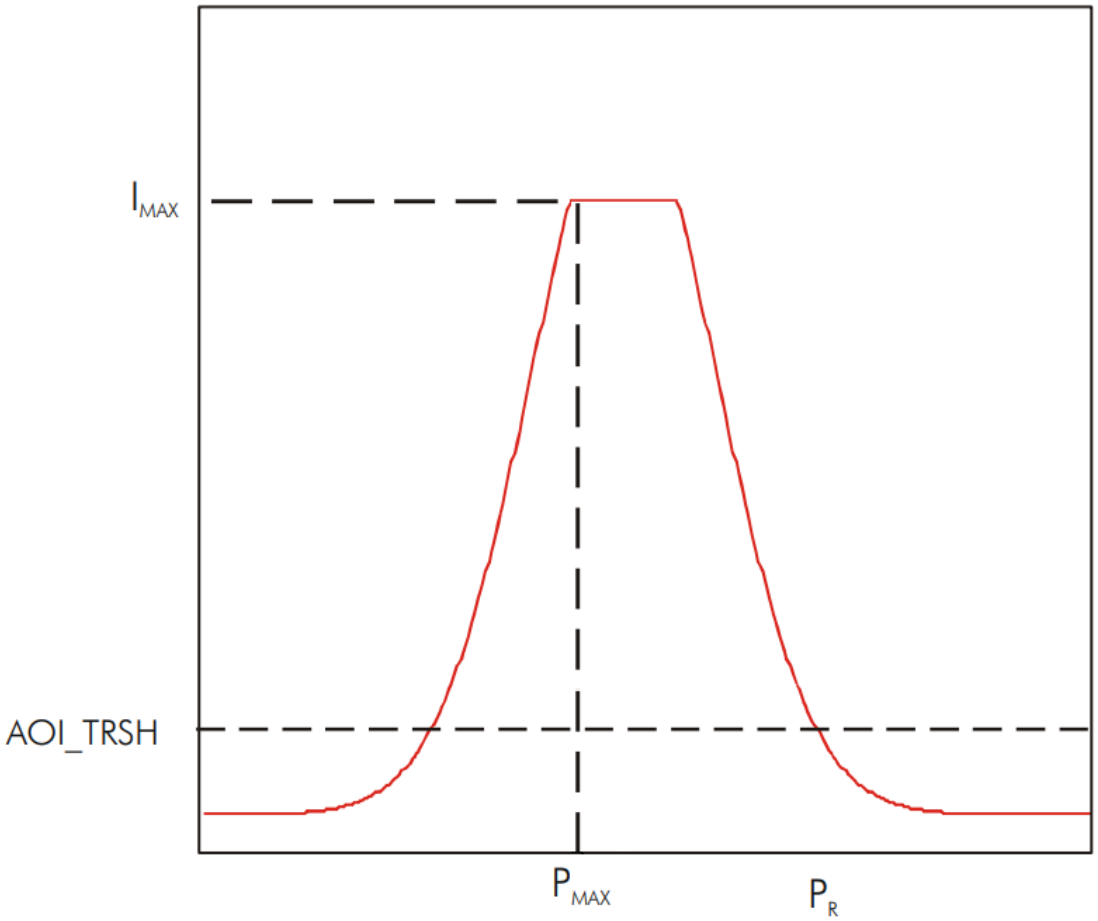


Figure 3.7: Saturation of optical sensor

3.1.4 Center of Gravity (CoG)

Compared to the previously mentioned algorithms, the CoG-algorithm is more intricate. The area between the curve and a set threshold is calculated by taking the sum of all the intensities I_p according to

$$I_s = \sum I_p \tag{3.3}$$

Furthermore, the sum of first order moment is calculated by multiplying the intensities I_p with the corresponding pixel-coordinates P

$$M_s = \sum I_p P \tag{3.4}$$

The pixel-coordinate corresponding to P_{COG} in Figure 3.8 can then be calculated by

$$P_{COG} = P_L + \frac{M_s}{I_s} \tag{3.5}$$

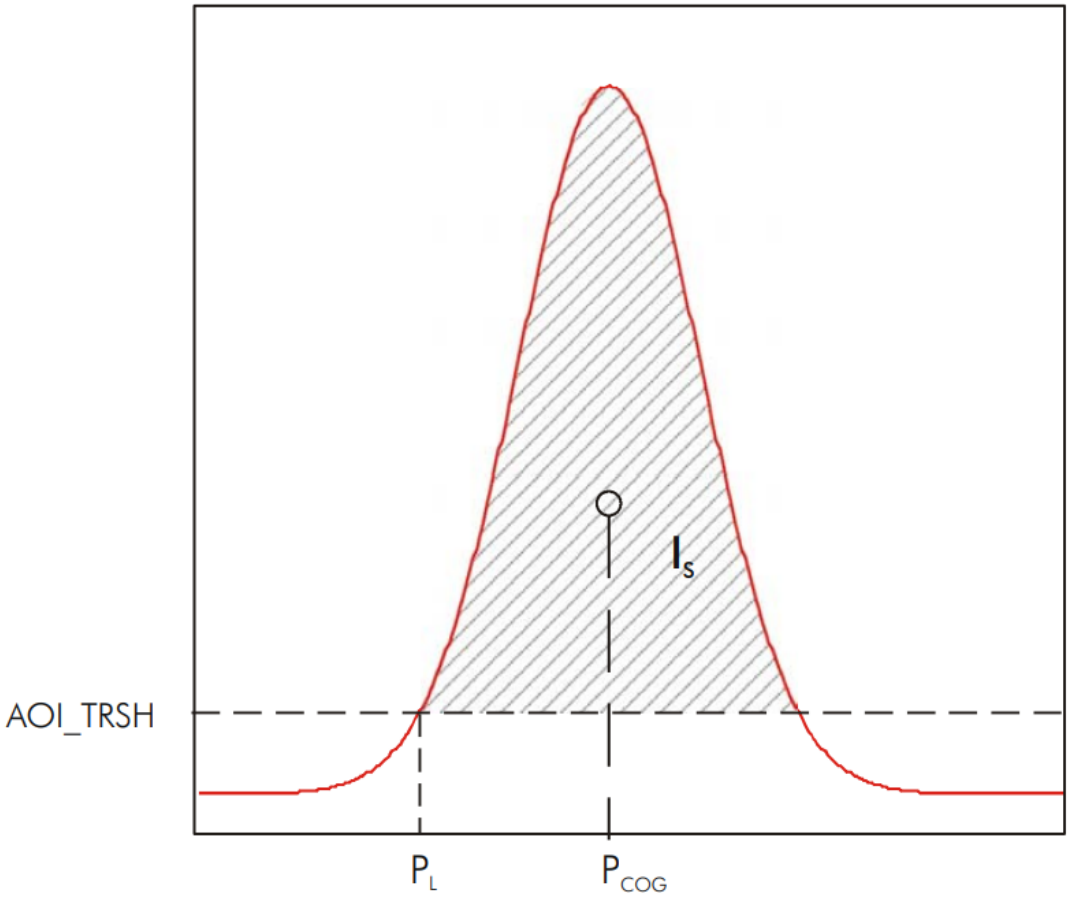


Figure 3.8: CoG-algorithm

By using this method, the camera can provide a resolution of up to 6 subpixel bits, which implies a resolution of $\frac{1}{2^6} \approx 0.0156$ pixels.

3.1.5 FIR-Peak

The last algorithm is FIR-Peak, which directly uses the FIR-filter with differential coefficients. The derivative of the intensity-curve is analyzed, where the extracted pixel-coordinate corresponds to where the derivative crosses the zero-value as shown in Figure 3.9. The resolution of this algorithm is the same as of the CoG-algorithm, providing a resolution of 6 subpixel bits.

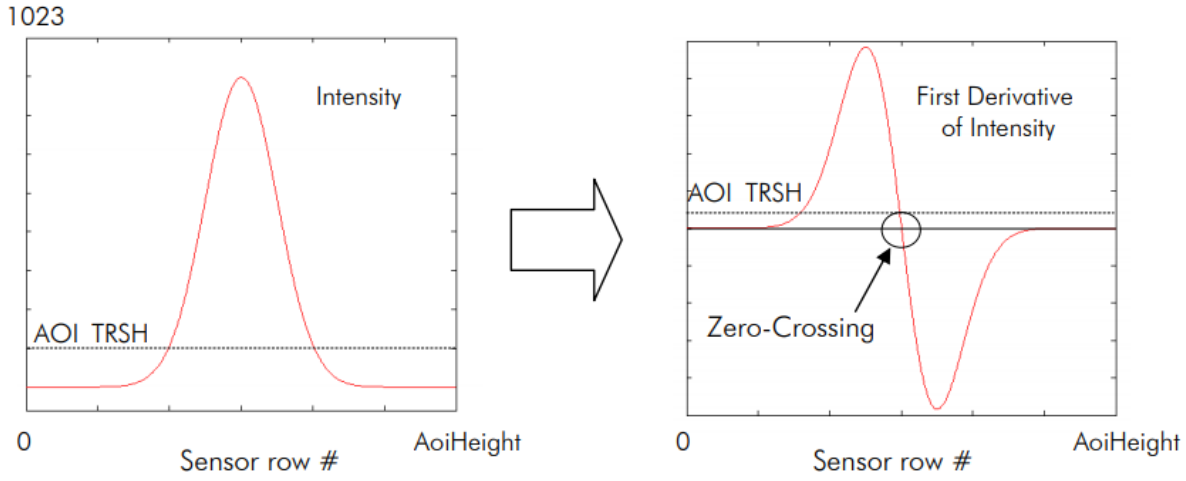


Figure 3.9: FIR-Peak Algorithm

As demonstrated in the previous sections, the camera has several available algorithms which is able to calculate the pixel-coordinate of the laser. However, based on the results of a previous report by the author [5], it was found that for high resolution reading with a smaller field of view – CoG mode is best suited. This is due to the fact that the width of the laser appear thicker on the image, which implies that the center of the width is a more accurate estimation of the laser position in the image. Based on the results of the stated report, CoG-mode is the algorithm used in the experiments of this thesis.

3.1.6 Multiple Slope Function

An additional feature of the C4-2040 GigE camera is the multiple slope function. By using this function it is possible to prevent saturation of the optical sensors. This may occur if the surface of the object is highly reflective. During the exposure time of the camera, the intensity-values of the pixels are monitored. If one of the cameras pixels reaches a set threshold of intensity within a certain fraction of the exposure time, the intensity of the pixel is restricted to rise for a set amount of time. As a result, pixels that would likely saturate within the exposure time are restricted from doing so.

This function would most likely be relevant if the experiments was carried out on an actual drill-pipe, as machined metal tend to be highly reflective. As the experiments in this thesis was restricted to 3D-printed objects, this function was not utilized. However, as it may be a deciding factor if the system is to be used on machined drill-pipes – the author decided to draw attention to this function such that it could be mentioned in the discussion.

3.1.7 Camera communication via Python

As it was desired to configure and communicate with the camera in real-time using Python, additional software had to be acquired. In this regard, the manufacturer of the camera was contacted and they were able to provide a standalone Python software development kit (SDK). This kit included the functions needed to communicate with the camera, and made it possible to both configure the camera and extract its readings of pixel-coordinates in real time by the use of Python programming. However, since the SDK

contained several functions which was not needed in this context, the necessary functions gathered in a common Python-script by Trym D. Hauglund. By using this script, the functions could be imported from a single file instead of several. For later reference, the function called *snap(hDev)* was the function which retrieved the image or pixel-coordinates, depending on the configuration of the camera. Due to the sheer size of the script, it is not attached in the appendix.

3.1.8 Lens

The lens mounted on the camera during the experiments was a Fujinon CF12.5HA-1 shown in Figure 3.10. The lens was advertised as a low-distortion lens designed for high resolution images. Additionally, the iris was equipped with locking knobs which reduced the chance of accidentally altering the intrinsic parameters after camera calibration.

Figure 3.10: Fujinon Lens,
Source: BHphotovideo

3.2 Laser

The laserline used in the experiments of this thesis was a 25mW Z-LASER Z25M18S3-F-640-LP45 with a wavelength of 640 nm. The laser was equipped with a Powell lens which delivers a uniform intensity along the laserline [24]. Another option would be the Gaussian lens, which intensifies the beam in the center of the laserline – this was however not required in the experiments. Additionally, the fan-angle of the laser was 45°, and contributed with a long laserline even on short distances from the object. The laser had the safety class 2M which meant that it was not required to use goggles while operating the laser. However, protective eye-wear were used as a precaution while operating the laser.

3.3 Triangulation configuration

In order to set the camera and laser projector in a permanent configuration, they were mounted on a rail as shown in Figure 3.11. The rail was designed by Trym D. Hauglund, and was configured in a

fashion that could be used in both of our theses. The configuration shown in the figure is referred to as "Standard geometry", and is one of the most commonly utilized configurations due to its accuracy and simplicity [25]. By using this configuration it is implied that the laser is aligned with the object that is to be scanned, and the camera views from an angle. Note that in the experiments the setup in Figure 3.11 was in a standing position, where the laser was positioned at the top and emitted a horizontal line.

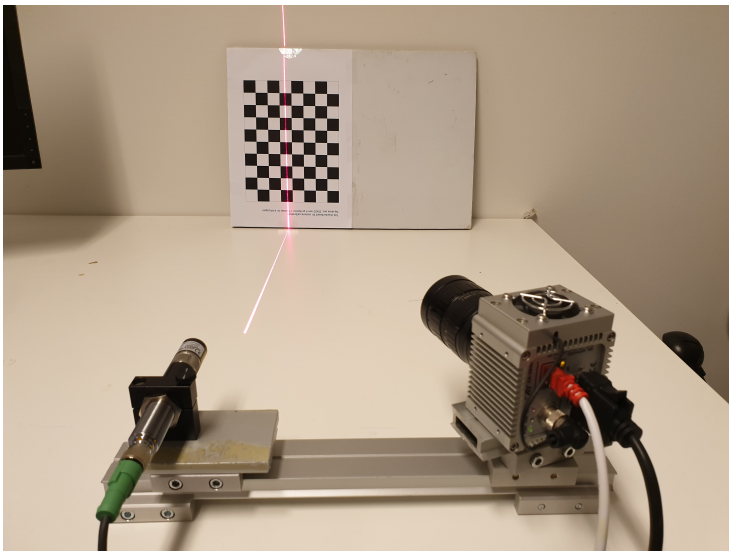


Figure 3.11: Camera/Laser configuration

Moreover, the distance between the laser projector and the camera was crudely measured to be approximately 20 cm, whereas the angle between them was measured using a goniometer to be in the region of 20°.

The resolution of the configuration depends mainly on two parameters, the field of view of the camera and the angle between the camera and the laser. As the laser appears as a horizontal line in the camera frame, it is the horizontal FOV which is relevant in this context. The horizontal FOV of the camera can be calculated using the following formula, which is stated by the manufacturer of the lens [26]

$$FOV = Y^{'}\frac{L}{f} \tag{3.6}$$

The parameter L is the distance to the object in focus, and f is the focal length of the camera. Moreover, the parameter $Y^{'}$ is the metric size of the image. This can be calculated by multiplying the number of pixels on the horizontal direction of the camera, with the size of the pixel. For the camera used in this thesis, the pixel-size was stated by the manufacturer to be 5.5µm [21], both vertical and horizontally. In addition, since there are 2048 pixels located in the cameras horizontal direction, the metric image size is calculated as

$$Y^{'} = \rho \mathrm{N} = 5.5\mu m \times 2048 = 11.264mm$$

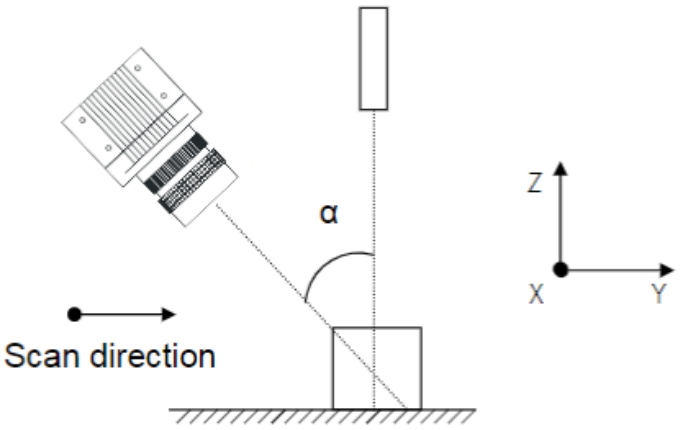


Figure 3.12: Camera/Laser configuration, Source : AT

From figure 3.12, the following resolution in the x-direction is stated by AT,

$$\Delta x = \frac{FOV}{\mathrm{N}}$$

By inserting (3.6), the resolution in x-direction becomes,

$$\Delta x = \frac{Y^{'} L}{\mathrm{N} f}$$

$$\Delta x = \frac{\rho \mathrm{N} L}{\mathrm{N} f}$$

Which results in,

$$\Delta x = \frac{\rho L}{f} \tag{3.7}$$

Furthermore, the resolution in the z-direction is given to be

$$\Delta z = \frac{\Delta x}{\sin \alpha} \tag{3.8}$$

As the equation states, the resolution in the z-direction is directly affected by the triangulation angle. One might argue that a large angle is preferred in order to achieve a high resolution. However, by increasing the angle also increases the risk of the laserline being hidden behind the geometry of the object. This effect is called occlusion and is highly undesired as it makes the camera unable to locate the position of the laserline. Furthermore, the resolution for both directions results in a metric value that represent how small of a change the camera is able to detect in the given direction. Consequently, a smaller number is preferred over a large one.

3.4 Laserplane calibration

In order to triangulate the pixels into spatial coordinates, the laserplane had to be represented as a plane in the camera frame. There are established methods of doing so, some of which includes scanning objects of known geometry [27]. However, in order to use the software and equipment already

at hand – another method was used in the laserplane calibration of this thesis. The method was found in an article on a related subject, which required only a calibration pattern mounted on a flat surface [28].

The proposed method was to capture images of a calibration pattern which allowed both its position and orientation to be calculated. If both the position and orientation of the pattern is known, the surface of the pattern can be expressed as a plane using point-normal notation. Furthermore, if the calibration pattern is positioned such that the laserline intersects the pattern, the pixel-coordinates coordinates of the laser can be triangulated with the plane of the calibration pattern. As this process results in three dimensional points located on the laserplane, a pointcloud can be generated by repeating this process using different positions and orientations of the calibration pattern. Assuming that the pointcloud consists of sufficient points, the parameters of the plane can be estimated by applying plane-fitting methods to the pointcloud. In this regard, it is preferred to capture images where the calibration pattern varies in the direction of the laser, as illustrated in Figure 3.13. The varying depth results in a better estimation, as the points yields a better representation of the span of the plane.

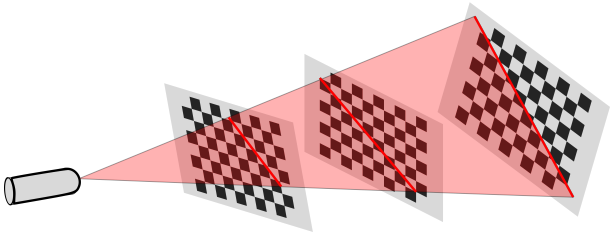


Figure 3.13: Laser intersecting calibration pattern

3.5 Angular measurement

As the intended scanning system relies on a real-time measurement of the pipe's orientation, a system had to be designed in order to extract these measurements during the experiments. First, a suitable sensor was to be acquired, which then would set the basis for the rest of the system in terms of requirements.

The sensors which was considered was two types of rotary encoders, absolute and incremental. Incremental encoders usually tend to be cheaper, but depending on their design, they only are able to tell the changes of orientation. A typical optical incremental encoder consists of a encoder-disk and two optical receivers [29]. The disk consist of a series of holes on its periphery, which enables light to pass to the optical receivers. Since the pulse generated by the receivers are 90 degrees out of phase, a sense of direction of the rotation can be derived. Furthermore, some of these encoders are equipped with a third sensor, which is dedicated for the *index pulse*. This pulse is generated once for each full rotation of the encoder-disk, and can be used as a reference to calculate an absolute angle measurement. However, as the encoder is powered, the encoder needs to locate the orientation which produces the index-pulse in order to provide an absolute reading.

An absolute encoder on the other hand, does not require to be rotated on start-up in order to provide the absolute angle. Depending on the resolution, each angle produces an unique signal produced by the sensory system. As a result, the encoder is able to produce an absolute angular reading the instant it is connected to power. In order to make the system more robust and suited for real-time application, it was decided that an absolute rotary encoder was more appropriate for providing the measurement.

3.5.1 Rotary encoder

Figure 3.14: Rotary encoder,
Source: Contelec

The acquired sensor was a Vert-X 28 rotary encoder manufactured by Contelec. The encoder allowed for an absolute measurement of the orientation by utilizing the magnetic Hall effect [30]. According to the documentation of the sensor [31], it required an input voltage of 5±0.5V, which would yield an analog output signal ranging from 10%-90% of the input voltage. Ideally, this would imply the sensor output ranging between 0.5-4.5 V for an input voltage of 5 V.

As the resolution of the encoder was stated to be 12 bits, one revolution of 360° would be divided into 2^{12} counts, which would imply that the sensor had a resolution of

$$\frac{360^\circ}{2^{12}\text{ counts}} = \frac{360^\circ}{4096\text{ counts}} = 0.0878^\circ \text{ per count} \approx 0.0015324 \text{ rad per count}$$

3.5.2 Arduino Uno

The Arduino Uno is an open-source development board which enable the users to design and build circuits based on inputs and outputs. In addition to being low-cost, the Uno is an easy-to-use alternative opposed to manually programming microcontrollers [32]. The board is equipped with an I/O interface in the form of both analog and digital input and output pins. Due to the product being open-source, there is a wide span of available libraries developed by the community which further expands the compatibility of the device.

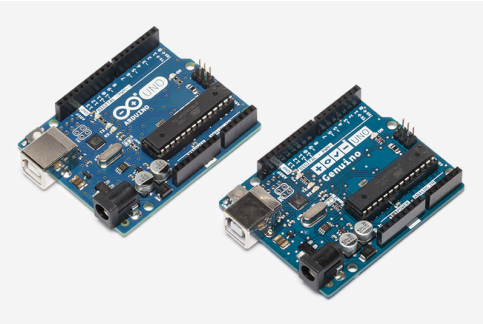


Figure 3.15: Arduino Uno,
Source: Arduino.cc

As the Arduino allowed for real-time communication via an USB-cable, the reading from the rotary encoder would be directly available during the scanning process. In addition, as the board was equipped with a 5V power supply, the rotary encoder could be directly supplied its required voltage without any additional circuitry.

3.5.3 Arduino programming

The programming of the Arduino was performed by using the Arduino integrated development environment (IDE), which allows programs to be written, compiled and uploaded to the Arduino. As seen in Figure 3.16, there are two main parts of the IDE, namely the *setup*, and the *loop*. The *setup* is run only once by the Arduino and defines which pins are used as input and output and/or which libraries are utilized. The *loop* on the other hand, is repeated as long as the Arduino is connected to power. This will contain the code which calculates the orientation of the encoder and transmits the information to the computer.

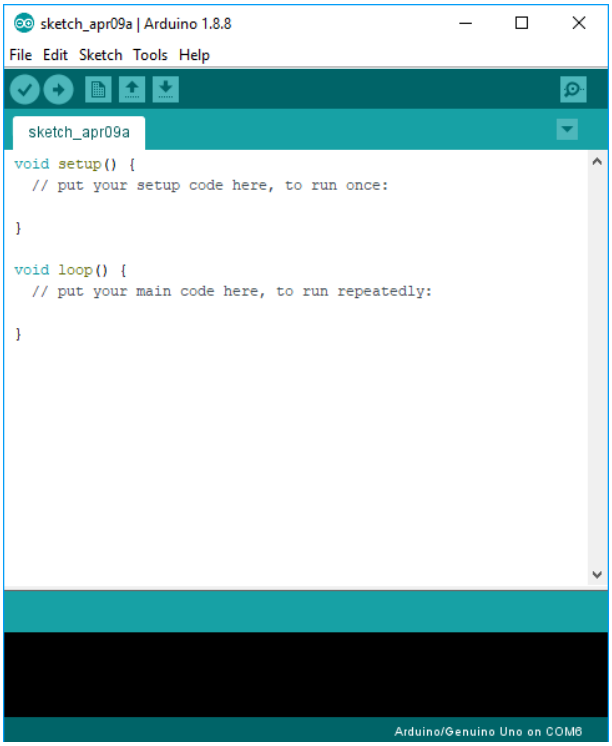


Figure 3.16: Arduino IDE

3.5.4 Arduino-Python communication

Since the camera used in the scanning process was communicating through Python, code was needed to be written in order to grant Python access to the angular reading from the Arduino. To achieve this, the scripts was supplied with the PySerial-module [33], which enabled direct communication to the port in which the Arduino was connected. As the Arduino was sending data through USB-port, this data could then be interpreted by the PySerial-module and further processed. Much time was spent trying to provide real-time reading of the angle, but did not result in a reliable solution. With the use of the Pyserial-module and a simple for-loop which read from the serial-port, Python was able to collect the angle seemingly without any problems. However, when a delay was added into the for-loop to simulate computation time, a large latency between the angular readings and the actual orientation occurred. This was later identified as a buffer problem, where the Arduino filled the serial buffer with readings at a higher rate then they were read by Python. This caused the buffer to be filled with outdated values, and Python would collect the oldest value in the buffer for each iteration.

However, after searching online for a potential solution, a post on the Arduino forums was found which solved the exact issue. A user by the name "Robin2" had solved the issue by using some clever coding [34]. Instead of making the Arduino produce the readings in a continuous loop, the Arduino only sent the reading when the function *CollectAngle* was called in Python (Appendix 7, line 50). When the function was called, a string on the format <'String'> was sent to the Arduino via the serial-port established by Pyserial. The Arduino remained idle until it received the start marker "<" of the string followed by the end marker ">". As seen in appendix 6 on line 23, when the end marker is received, the boolean *newData* is set to True. This initializes the function *replyToPython* on line 35 which sends the string back to Python containing the angular measurement.

Moreover, immediately after Python sends the string requesting the reading to the Arduino, the function *recvLikeArduino* starts reading the serial-port for the reply. The reply is again identified with the start marker "<", in which each received character is appended to a string until the character ">" is read. As the string is completed, the value of the reading is collected by a split of the string with spaces as the separator. As the string sent from the Arduino has the format "Angular reading: 99.61", the split of the string will be a list on the form [Angular,reading:,99.61]. The angular reading will then be the third element of the list, which corresponds to the index two as shown line 56 in appendix 7. Unlike Matlab, in Python the first element in a list or array has index 0, hence index two for the third element. The code found was mainly used as is, but slightly modified in order to read the angle of the encoder.

3.6 Dedicated triangulation function

In order to more easily triangulate the readings from the camera during the experiments, a function was written which triangulated the given pixels based on the given plane. This function composed of three sections, where the first two is preparing the pixel-coordinates for the triangulation process. First, the array of pixels were cleaned of pixel-coordinates that had a y-value equal to zero. These coordinates represented the pixel-columns where the laser was not present, and since they would be undistorted in the same fashion as the others, these coordinates had to be removed prior to the triangulation. This was simply done by iterating through the y-values of the pixel-array, and generating a new array with pixel-coordinates which had a non-zero y-value. Next, the pixel-coordinates had to be undistorted and transferred to the camera-frame. This was done by using OpenCV's *undistortPoints*-function, which undistorted pixel-coordinates according to (2.30) and (2.31). The undistorted pixel-coordinates were then transformed to metric coordinates in the camera frame by using (2.25).

In order to perform this operation, the function needed access to the camera-matrix and the distortion coefficients which was explained in section 2.7. These parameters were loaded by calling the *loadCali-Param*-function described in section 2.7.3.

Lastly, the undistorted coordinates are triangulated with the desired plane using the theory of line-

plane intersection from section 2.9. As the estimated planes were to be saved locally as .npy-files (Numpy array file), the desired plane was used by loading the corresponding file containing its point-normal representation. Moreover, the triangulated points were appended to an array called *ext_points*, which was the output of the function. The finished function is shown in appendix 13.

3.7 Scanning program

As the readings of the camera needed to be related to a specific orientation of the object, a program was written which combined the two. where By establishing contact with both the camera and the Arduino, both the extracted pixels and angular reading was available for the program. Furthermore, the program was designed with a while-loop which was not terminated until the key 'q' was hit on the keyboard. For each iteration of the loop, a CoG-reading was performed by the camera as the *snap(hDev)*-function was called. In the same iteration, the function *CollectAngle()* from section 3.5.4 was called, which sent a request to the Arduino for the encoders orientation. The pixel-coordinates from the camera and the orientation from the Arduino was then arranged in a dictionary generated by the program. The dictionary was appended by using the angle as key and the pixels-coordinates as the corresponding value – generating a dictionary on the following form.

$$\text{Dictionary} : [\text{Angle}_1 : \text{Pixel-coordinates}_1, \text{Angle}_2 : \text{Pixel-coordinates}_2 \ldots \text{Angle}_n : \text{Pixel-coordinates}_n]$$

Initially, the pixels were triangulated into spatial coordinates before it was appended to the dictionary with its corresponding angle. However, this resulted in that one iteration of the function took approximately 0.2 seconds to complete. In order to achieve a higher frequency of readings, it was decided that the pixel-coordinates provided from the camera would not be triangulated during the scan. Instead, the pixel-coordinates was stored as is, such that they could be triangulated after the scanning was finished. This modification reduced the elapsed time of one iteration to approximately 0.1 seconds. Furthermore, as the 'q'-button was pressed to terminate the scanning process – the dictionary from the process was saved as a .npy-file in a desired folder.

4. Experimental results

In this chapter, all the experiments throughout the thesis are explained together with their result. In order to perform a scan of an object which relates its curvature to the angle of orientation, a method of measuring its orientation must be made available. As a result, the first part of this chapter is to design a system which ensures an accurate measurement of the angular orientation of the object as it is being scanned.

Next, a laserplane calibration is to be performed such that the relationship between the camera and laserplane can be accurately estimated. In this thesis, this relationship is estimated by generating a pointcloud of points located on the laserplane. The laserplane is then estimated from the pointcloud using three plane fitting methods. A test is then carried out in order to determine the accuracy of the three different plane-estimations.

Lastly, a 3D-printed cylindrical shape is scanned in a manner that combines the angular measurements with the pixel-coordinates from the camera. This information is then presented in two different ways. First, the readings are used to reconstruct the shape of the scanned object. Next, the readings are represented as a function of the objects orientation which is considered to be a better representation with regards to supplying a welding robot with positional input. The chapter is then concluded with a suggestion to how the data can be treated to calculate the path of a robot which will follow the curvature of the rotating object.

4.1 Angle measurement

In order to relate the readings from the camera to the different orientations of the object, the angle of the object had to be precisely measured. Initially, the circuit which was supposed to provide the angular readings was put together as simply as possible. As the Arduino was able to provide the voltage required by the rotary encoder, the encoder was simply connected to Arduino according to Figure 4.1. The analog output of the sensor was connected to analog port "A0" on the board, and simple code-snippet was written in the Arduino IDE which allowed easy reading of the analog input.

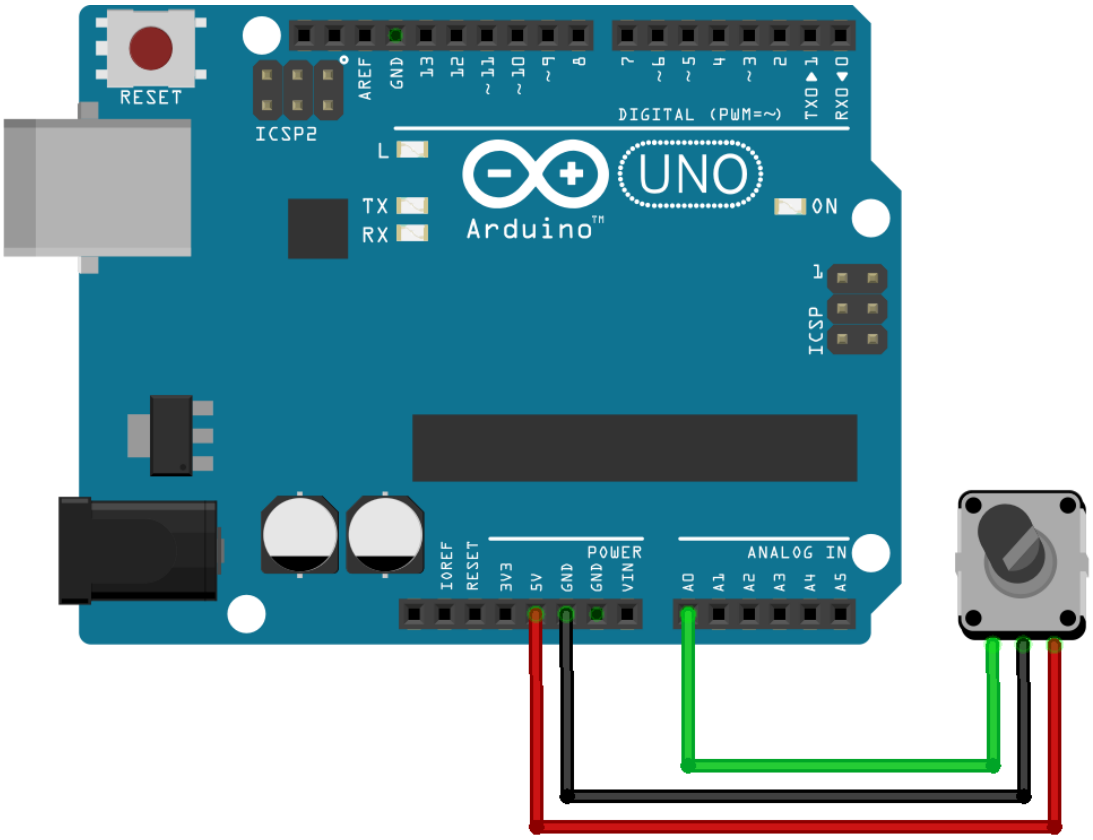


Figure 4.1: Arduino with directly connected rotary encoder

When inspecting the value of the analog port, it was noticed that the registered value only ranging from 104 to 921 as the encoder was rotated a full revolution. As the values 101 and 902 represented the span of angles between 0° and 360°, this implied that the resolution of the angle was approximately

$$\frac{360^\circ}{921 - 104} = \frac{360^\circ}{817} \approx 0.44^\circ$$

As the resolution of the encoder was stated to be 12-bit (i.e 4096 counts), the span of the counts was expected to be larger. Upon further investigation and reading the Arduino's manual, it was discovered that the Arduino Uno's internal analog/digital converter (ADC) only had 10 bits (i.e 1024 counts) distributed over a 5V range. Since the loss of resolution between the sensor and Arduino was substantial, further improvements had to be made to the circuit.

4.1.1 Analog/Digital converter

There are several other Arduino-boards available that supports 12-bit resolution on their on-board ADC's [35]. However, it was not desired to acquire a second Arduino for this purpose. Instead, in order to increase the resolution of the readings, an external 12-bit ADC was purchased which could provide the Arduino with a digital signal. The acquired ADC was an ADS1015 by Adafruit, which was directly compatible with Arduino programming, as it included a library with all the functions needed for it to operate. Furthermore, the ADC was able to operate on both 3.3V or 5V and supported up to four separate analog input-signals (A0 - A3). Moreover, the voltage of the input-signals was required to be within the ground potential and input voltage (VDD).

Figure 4.2: ADS1015 Analog/Digital converter,
Source: Elfa Distrelec

As seen in image 4.2 the ADC has 4 input-pins named A0 to A3.

In reality, the ADS1015 has an 11-bit resolution – but with an additional sign-bit [36]. As the 11 bits are able to produce 2048 counts, the sign-bit allows the same amount of negative counts, resulting in a total of 4096 bits, or 12-bit resolution.

Moreover, there was two different connection modes supported by the ADS1015, namely *Single Ended*- and *Differential*-mode.

4.1.1.1 Single Ended Mode

As the name might suggest, this mode only requires one of the input pins of the ADS1015 which allows more sensors connected to one single ADC. Using this mode, the difference between the input-pin and ground is evaluated and converted, according to (4.1).

$$\Delta V = V_{Ground} - V_{input} \tag{4.1}$$

However, as the ADC did not allow negative input-voltages the difference between the two was restricted to only positive values, assuming ground is zero volts. As a result, the counts located in the negative interval is not utilized – which limited the resolution to only half.

4.1.1.2 Differential Mode

In comparison to single ended mode, differential mode occupies two input-pins per sensor. Instead of comparing the signal from the sensor to ground, the signal is compared to a reference voltage instead, given by (4.2).

$$\Delta V = V_{ref} - V_{input} \tag{4.2}$$

By choosing the reference voltage with thought, this mode can utilize the negative counts as well. Instances where $V_{input} > V_{ref}$ will result in a negative difference, which would be mapped to the negative signed bits. Additionally, since it is the difference between the two pins that is considered – the differential mode is more robust towards common noise than the single ended mode.

4.1.1.3 Input Gain

The ADC had the possibility to amplify the input signal, in order to set the appropriate scaling of counts. Depending on the amplitude of the a connected sensor, it is possible to adjust the gain of the

ADC accordingly. The default gain of the ADS1015 was 2/3, which would distribute the 4096 counts on a interval of ± 6.144V where each count represent a change of 3mV. Table 4.1 shows the different gains supported by the ADS1015, in addition to the different voltage intervals and count values.

Suppose a sensor has a output from 0-1V and is connected to the ADC in differential mode with a reference-voltage of 0.5V. The difference between the two would range between ±0.5V, which with the default would use the 3mV per count scaling. As a result, the interval from -0.5V to 0.5V would be distributed over approximately 333 counts. The appropriate scaling would for this example be 8, which would divide the 1V interval in segments of 0.25mV for a total of 4000 counts.

Table 4.1: Gain parameters

Gain	**Voltage Interval**	**Count value**
2/3	±6.144V	3mV
1	±4.096V	2mV
2	±2.048V	1mV
4	±1.024V	0.5mV
8	±0.512V	0.25mV
16	±0.256	0.125mV

4.1.2 Circuit design

Due to the restriction of input-voltage to the ADC, the ADC was connected to the 5V supply on the Arduino. This ensured that the analog input from the encoder, which was in between 0.5-4.5V, would not exceed input-voltage of the ADC. Furthermore, as the purpose of the ADC was to increase the resolution of the readings – it was decided that the encoder would be connected in differential-mode with the use of a reference voltage. As the encoder voltage spanned 4V, the reference voltage was decided to be 2.5V. According to (4.2), the voltage difference would be

$$\Delta V = \begin{cases} -2, & \text{if } V_{input} = 4.5V \\ 2, & \text{if } V_{input} = 0.5V \end{cases} \quad , \qquad \Delta V \in [-2, 2] \tag{4.3}$$

Furthermore, the gain had to be adjusted accordingly to the expected voltage difference. As using the standard gain would result in loss of resolution, the appropriate gain was decided to be 2. Choosing this gain distributed the counts over the voltage interval ±2.048, as shown in table 4.1.

4.1.2.1 Reference voltage

In order to generate a reference voltage a simple voltage divider was designed by using a 10k trimpotmeter. The trimpotmeter was connected between the Arduino's 5V port and ground, as shown by Figure 4.3. Despite the figure, the trimpotmeter is a single component with a variable resistance. The right setting of the trimpotmeter was found by probing the output with an oscilloscope, and adjusting the resistance until the desired voltage was achieved.

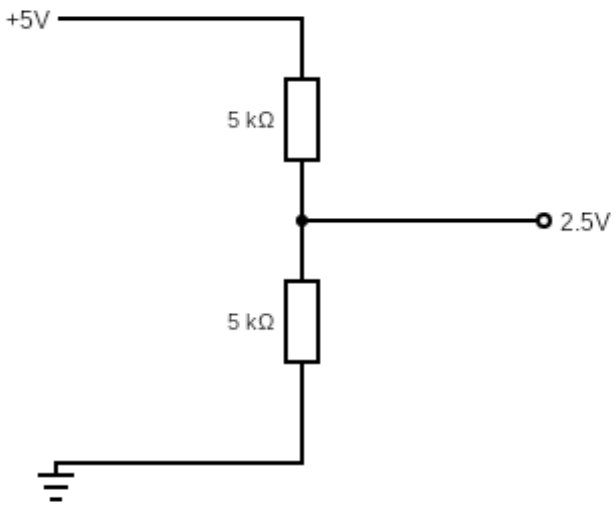


Figure 4.3: 2.5V reference voltage using trimpotmeter

Furthermore, to allow for easy connection of the ADC to the breadboard – header pins was soldered onto the chip.

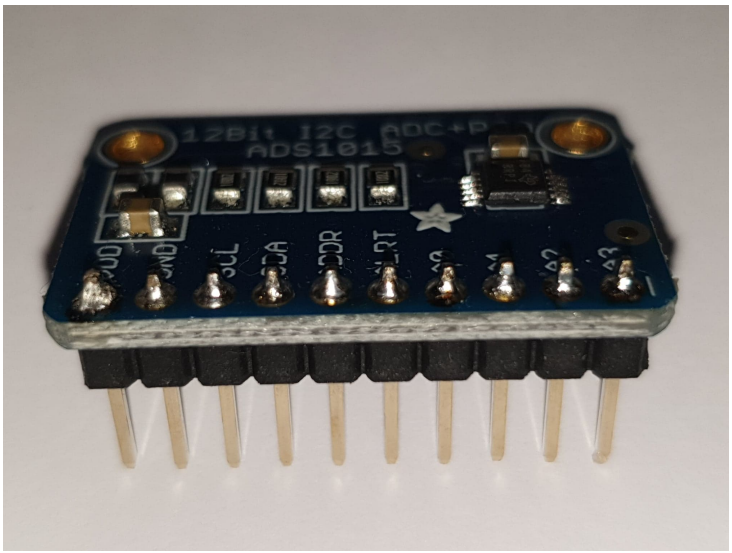


Figure 4.4: Headers on ADS1015

The finished schematic of the circuit is shown in Figure 4.5, followed by an image of the actual circuit in Figure 4.6. Note that the color of the wires between Figure 4.5 and Figure 4.6 does not necessarily match.

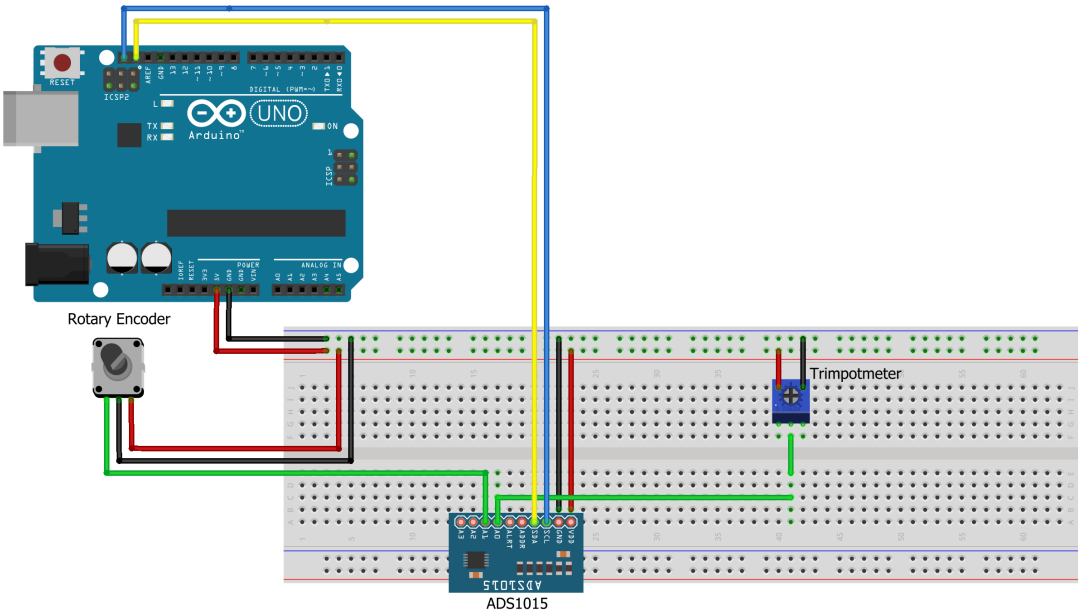


Figure 4.5: Schematic of Arduino circuit

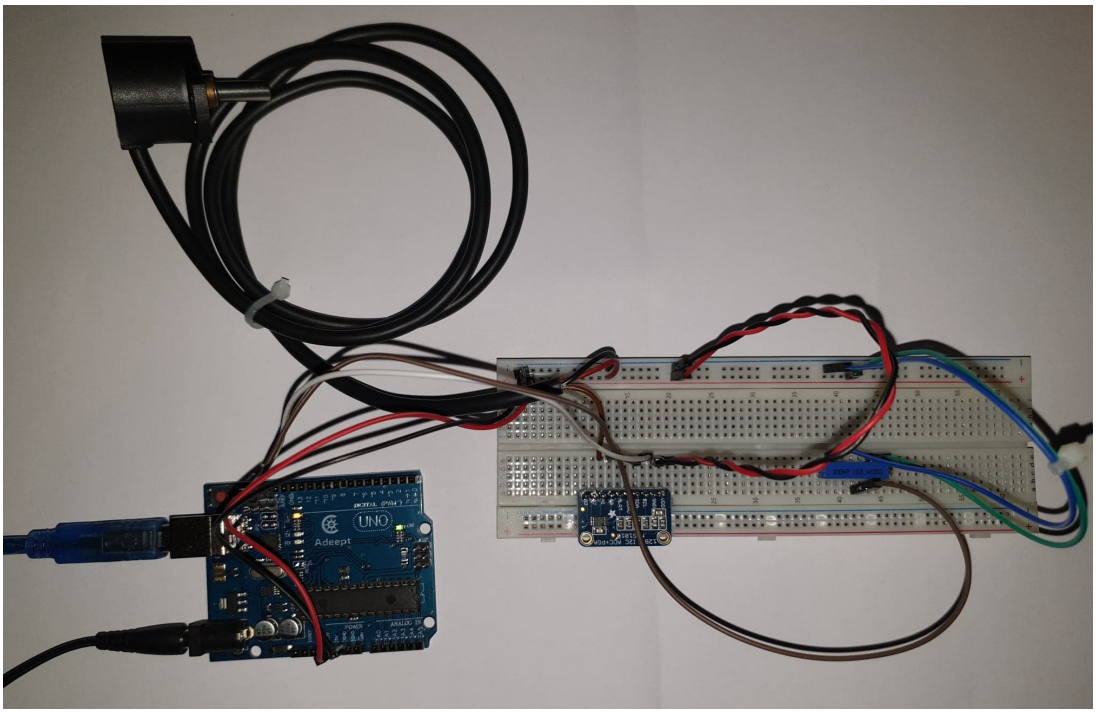


Figure 4.6: Photo of Arduino circuit

As previously done with only the sensor connected, a simple code-snippet was written in order to see which values the Arduino received from the ADS1015. As the encoder was turned a full revolution, the counts ranged between -1963 to 1970 for a total of 3933 counts. From these values, the resolution of the angle per count was estimated to be

$$\frac{360^\circ}{1970 - (-1963)} = \frac{360^\circ}{3933} \approx 0.0915^\circ$$

Compared to the previous test where the smallest change in the angle that was detectable by the Arduino was close to 0.44°, the angle could now be determined with an accuracy of 0.0915°. However, it was noticed that the full potential of the resolution was still not reached. As the gain used corresponded to a 1mV span of each count, a voltage interval of 4V would ideally result in a total count of 4000. This was a due to the Arduino's inability to provide exactly 5V. The voltage of the port was measured to be approximately 4.915V, which was a 0.085V deviation from the expected voltage. However, as the deviation in voltage only resulted in a loss of resolution of about 0.0015 °, additional changes was not done to the circuit.

Furthermore, a test was carried out where both the analog signal from the sensor and the digital signal from the ADC was logged during rotation of the encoder. By using the function *map* in the Arduino IDE, the range of counts for the two signals were mapped to values between 0 and 359.99 which represented the angles. As this function was only to map one range of integers to another, the angle 359.99 was written as 35999 and later divided by 100. The code which was used is shown in appendix 4.

Figure 4.7 shows the values registered by both signals during and arbitrary rotation of the encoder, and it demonstrates that both readings tends toward the same angle.

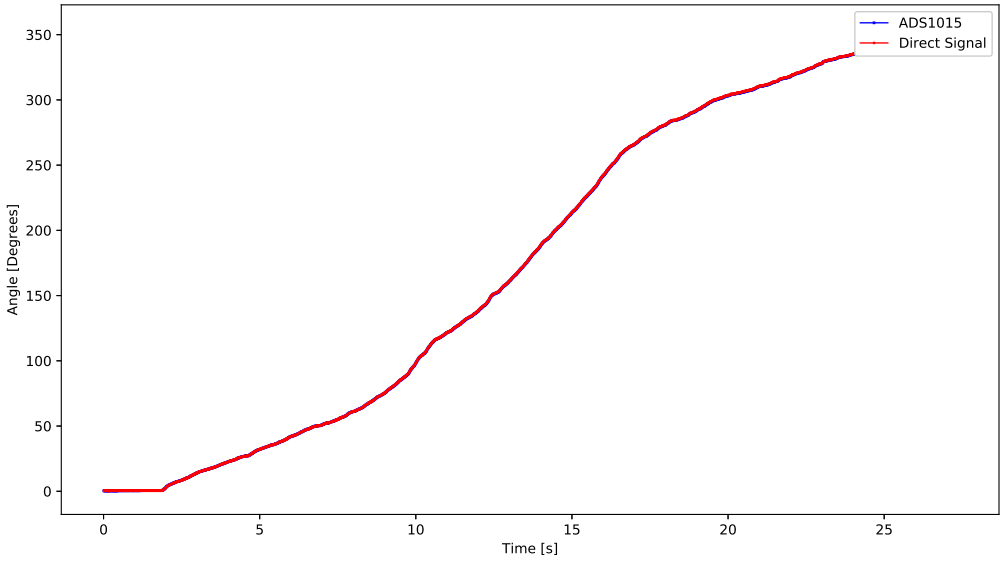


Figure 4.7: Comparison between analog and digital reading

However, if the figure is enhanced, the difference between the readings becomes more apparent. Figure 4.8 demonstrates that the analog signal changes its value in a much cruder fashion than the one from the ADS1015. This is naturally a direct result of the difference in resolution.

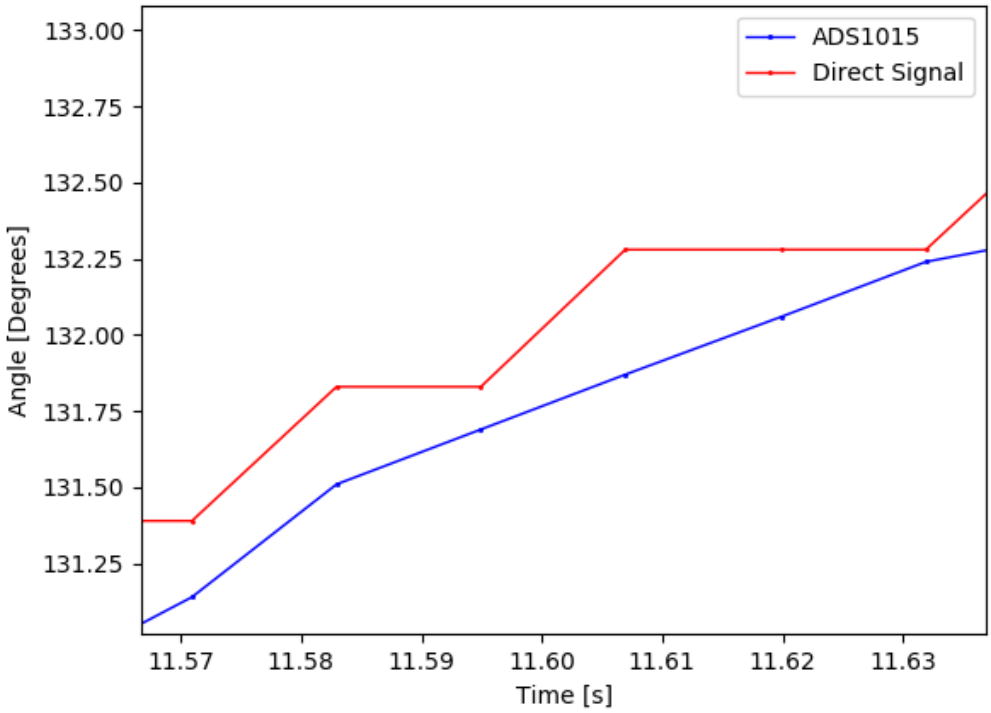


Figure 4.8: Closer comparison between the analog and digital reading

4.2 Laserplane calibration

In order to triangulate the pixel-coordinates of the camera into spatial coordinates during the scanning process, the relationship between the laser and camera had to be determined. Following the proposed method described in section 3.4, images had to be taken of the checkerboard from which the orientation and position of the checkerboards could be calculated using the calibration software of Matlab. Furthermore, with the laserline being present on the checkerboard – the pixel-coordinates of the laserline could be identified by performing a intensity reading of the same checkerboard with the same orientation and position. Combining this information would make it possible to triangulate the pixel-coordinates of the laserline to the checkerboards plane, which would result in spatial coordinates in the camera frame. With enough spatial coordinates which was located on the laserplane, a pointcloud could be formed from where further analysis would estimate the parameters of the plane.

4.2.1 Data generation

In order to generate a pointcloud of the points located on the laserplane, sufficient points from the laserplane needed to be gathered. As mentioned in the introduction of this section, the data consisted of images of the laserline being present on the checkerboards with a corresponding intensity-reading. In order to collect both the images and intensity readings from a checkerboards pose, a program was written in collaboration with Trym D. Hauglund. For each pose, the program configured the camera into CoG-mode and performed an intensity reading. The intensity reading was formatted as an array of size (2048,2) which contained the pixel-coordinates of the laserline. The array was then saved as a .npy file, and the program proceeded to configure the camera to image-mode and an image was taken of the checkerboard. The captured image was saved as a .png file in the same folder as the array. However, as these two operations where asynchronous, the checkerboard was required to remain stationary while the program finished. If the checkerboard was relocated between the two operations, the readings would

result in a disparity and yield inaccurate results when the laserplane was to be estimated. Moreover, as there was no live videofeed, placing the checkerboards correctly in the cameraframe proved to be a time-consuming task. Many of the images taken had to be redone as parts of the checkerboard was not properly captured by the camera.

As the pose of the checkerboard had been captured using CoG- and image-mode, the checkerboard could be re-positioned and reoriented for another iteration of the program. Moreover, as it was necessary to relate the images to the corresponding array-files, the files had to be saved in a systematic manner. The images was named numerically, starting with the filename "1.png". Consequently, the corresponding array-file would be named "pixcoord_1.npy", and so on. The code for this program can be found in appendix 3.

For a total of 10 iterations, each pose of the checkerboard was analyzed first with the CoG-mode activated, followed by image-mode as described by the program. The captured images was then analyzed using the calibration software of Matlab in order to identify the checkerboards location and orientation in the camera frame as shown in Figure 4.9.

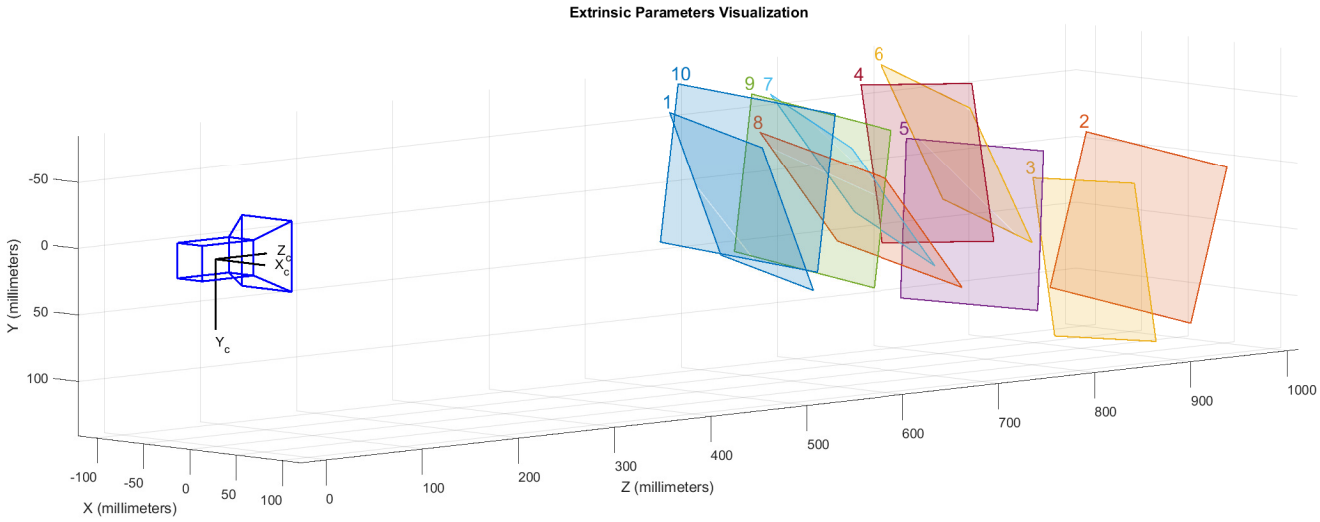


Figure 4.9: Checkerboard poses

Moreover, the checkerboard used had a pattern of 7x9 with square-size of 20mm. The size allowed the laserline to be present on the checkerboards for a larger variation of distances from the camera. This was desired as more readings from different distances gave more information regarding the direction of the laserplane. The square-size of the checkerboard was confirmed using a caliper, as a deviation would result in errors of the extrinsic values. In order to accommodate the size of the checkerboard, the focus of the lens was set to approximately 700 mm.

In addition, a proper camera calibration without the laser was carried out prior to this experiment – as the laserline could influence the estimation of the intrinsic parameters (i.e the camera-matrix and distortion coefficients).

An example of the analyzed images is shown in Figure 4.10, where the laserline can be seen present on the checkerboard. The checkerboard corresponds to number 4 in Figure 4.9.

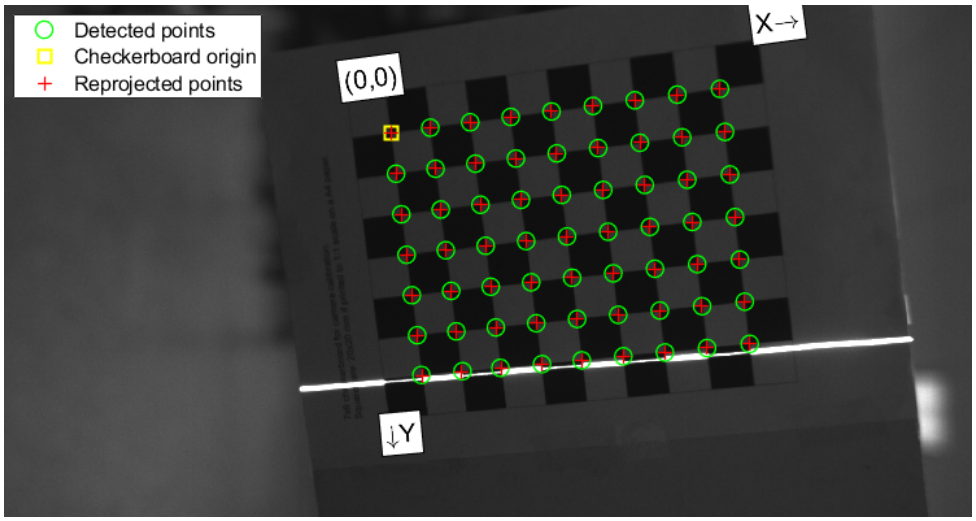


Figure 4.10: Laser on checkerboard

The rotation matrix and translational vector corresponding to the checkerboard in Figure 4.10 was given to be

$$R^{cam}_{checker} = \begin{bmatrix} 0.9716 & 0.1248 & 0.2009 \\ -0.1079 & 0.9898 & -0.09300 \\ -0.2105 & 0.0687 & 0.9752 \end{bmatrix} \quad t = \begin{bmatrix} -52.78 \\ -66.5694 \\ 720.931 \end{bmatrix} \tag{4.4}$$

Furthermore, the extracted pixel-coordinates from the corresponding iteration is shown in Figure 4.11. If compared to the actual image, Figure 4.10, the resemblance between the position of the laserline and extracted pixel coordinates are apparent.

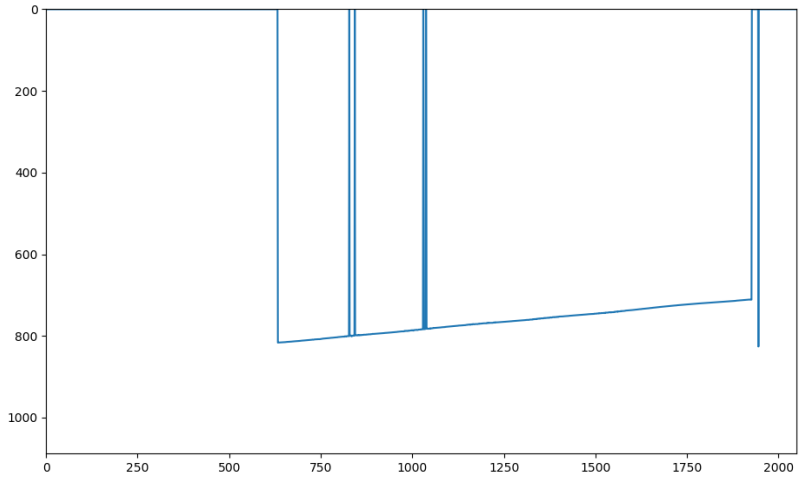


Figure 4.11: Extracted pixel-coordinates

It was noticed that the bright spot on the bottom right of Figure 4.10 was registered by the camera, which also can be seen from Figure 4.11. As these pixels where not a part of the laserline, they had to be removed from the data set to ensure correct estimation of the laserplane. As the camera was stationary during the imaging-process, the bright-spot had the same location throughout the images which meant the last pixels could simply be removed from the arrays.

4.2.2 Point cloud generation

By using the data collected in the previous section, it was possible to express the surface of the checkerboard as a plane in the camera-frame. Using point-normal notation, the translational vector which was the position of the checkerboards origin, represented a point on the plane. Furthermore, as each of the columns in the rotational matrix represented the unit directions of the checkerboards axes – the vector normal to the checkerboard could be directly extracted as being the last column of the matrix.

As shown in appendix 8, each of the .npy-files containing the extracted pixel-coordinates were loaded into the script. In addition, the function *loadCaliParam* from section 2.7.3 was called in order to gain access to the values gathered from the camera calibration. Due to systematic naming, the first .npy file that was treated corresponded to the first rotation matrix and translational vector collected by the *loadCaliParam* function. First, the function removed the faulty pixels-coordinates registered by the camera. Some of the valid data points was lost during this process, however this was preferred over having incorrect readings which was not located on the plane of the checkerboard. Furthermore, OpenCV's function *undistortPoints* was utilized in order to undistort the pixels in the array. The function is based on (2.30) and (2.31), and calculated the undistorted pixel-coordinates according to the given distortion-coefficients. Furthermore, by using (2.24), the function transforms the undistorted pixel-coordinates into metric coordinates in the camera-frame. The homogeneous transformation is assumed to be 4x4 identity matrix as an external coordinate system is not relevant in this context.

As the pixel-coordinates now was expressed in the camera-frame, they were converted to homogeneous form by adding a third dimension which had length 1, as shown in line 31 in appendix 8. As location and orientation was already expressed in the camera-frame, no transformation between frames was needed, which implied that the theory behind line-plane intersection from section 2.9 could then be directly applied.

From (2.44), p_0 could be set to be the origin of the checkerboard, and l_0 could set to be the origin of the cameras frame, $\mathbf{n}$ was the normal of the checkerboard and $\mathbf{l}$ was the unit direction of the normalized pixel-coordinate. The resulting value for d, was then the distance from the camera to the point on the checkerboard, in the direction $\mathbf{l}$. The spatial coordinate was then calculated by (2.42) and appended to an array dedicated for the extracted coordinates. Following the previous shown sample from Figure 4.10, with the values of (4.4) – the point-normal representation of the checkerboard was given by

$$\mathbf{n} = \begin{bmatrix} 0.2009 \\ -0.09300 \\ 0.9752 \end{bmatrix} \quad d = \begin{bmatrix} 52.78 \\ -66.5694 \\ 720.931 \end{bmatrix} \tag{4.5}$$

When the pixel-coordinates of Figure 4.11 was triangulated to the plane, the spatial coordinates of the laserline were calculated. The extracted coordinates for the laser on this checkerboard is shown in Figure 4.12.

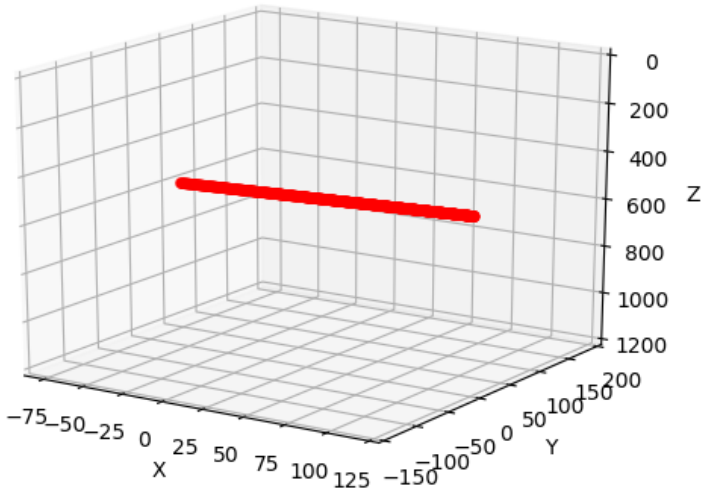
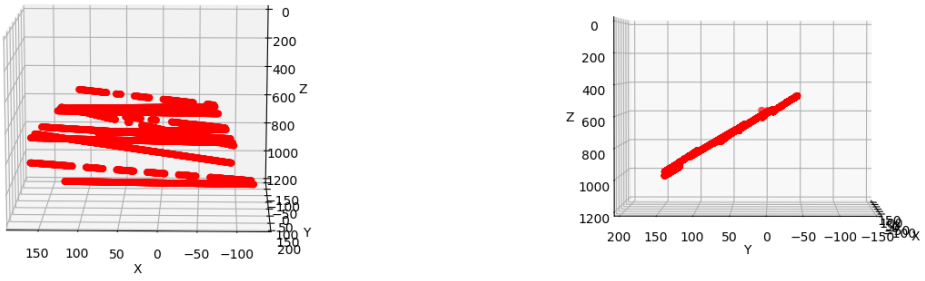


Figure 4.12: Spatial points of laserline

The procedure was repeated for all 10 images and corresponding .npy files, resulting in a pointcloud of 8818 individual points. These points formed the shape shown in Figure 4.13a. By inspecting Figure 4.13b, the shape of a plane is more apparent.



(a) Pointcloud front view

(b) Pointcloud side view

Figure 4.13: Generated pointcloud

4.2.3 Plane fitting

As the triangulation process in this thesis relied on the laserplane being expressed in the cameras frame – the pointcloud spanning the plane of the laser had to be further processed.

Three different methods where used in order to express the plane in the camera-frame. First a plane is estimated by expressing the plane as a linear system from which the parameters of the plane are estimated using matrix algebra. The two other methods is based on defining a cost function in which a optimization

problem is solved. The difference between the two latter methods is the use of weights. The weights was applied in an attempt to differentiate the uncertainty related to each of the points in the pointcloud.

4.2.3.1 Least squares plane estimation

As the pointcloud was available from the procedure in the previous section, this method of plane fitting required small amounts of coding in order to achieve a representation of the plane. The code is shown in appendix 9, and demonstrates the methods simplicity, as very few lines of code are needed. By applying the theory from section 2.10.1, the matrix **A** and vector **b** could be formed. By arranging the x, y and z-coordinates of the pointcloud, the system which expressed the plane could be formed.

$$\begin{bmatrix} x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots \\ x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} \tilde{A} \\ \tilde{B} \\ \tilde{D} \end{bmatrix} = \begin{bmatrix} -z_1 \\ \vdots \\ -z_n \end{bmatrix} \quad , \quad \mathrm{A}\mathbf{x} = \mathbf{b}$$

The A-matrix was then evaluated to calculate the Moore-Penrose matrix.

$$M = (\mathrm{A}^T\mathrm{A})^{-1}\mathrm{A}^T \tag{4.6}$$

By multiplying the M-matrix with the z-coordinates, the resulting b-vector containing the estimated coefficients was given as

$$\mathbf{x} = M\mathbf{b} = \begin{bmatrix} \tilde{A} \\ \tilde{B} \\ \tilde{D} \end{bmatrix} = \begin{bmatrix} -0.04915 \\ -2.60167 \\ -588.64971 \end{bmatrix} \tag{4.7}$$

By inserting the values of $\tilde{A}$,$\tilde{B}$ and $\tilde{D}$ into (2.48), the equation of the estimated plane became

$$-0.04915x - 2.60167y - 588.64971 = -z \tag{4.8}$$

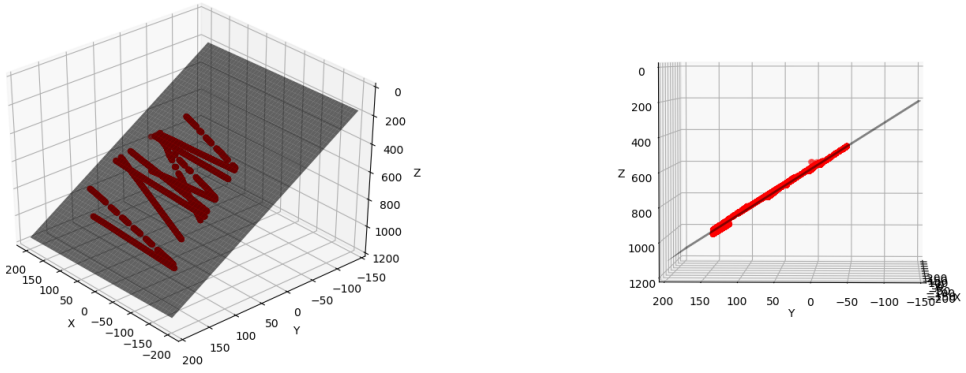
Or on standard form,

$$0.04915x + 2.60167y - z + 588.64971 = 0 \tag{4.9}$$

Moreover, since it was the point-normal representation that was going to be used in the triangulation process – the representation in 4.9 needed to be converted. By applying the theory from section 2.8, (4.9) was transformed to point-normal representation using the *planeify*-function described in section 2.8. The point-normal representation of the plane was then given by,

$$\mathbf{n} = \begin{bmatrix} 0.04915 \\ 2.601567 \\ -1 \end{bmatrix} \quad , \quad d = \begin{bmatrix} 0 \\ 0 \\ 588.64971 \end{bmatrix} \tag{4.10}$$

The estimated plane was plotted with the pointcloud to illustrate its fit. As shown in Figure 4.14, the plane was fitted to the apparent plane spanned by the pointcloud.



(a) LS-plane

(b) Side view

Figure 4.14: Least squares plane representation

4.2.3.2 Optimization using orthogonal distances

As described in section 2.10.2, the plane could be fitted to the pointcloud by considering the orthogonal distances between the points and the fitted plane by evaluating the following cost-function.

$$\begin{aligned} &\min_{\mathbf{n}} \sum_{i=1}^{N} ((p_i - c)^T \cdot \mathbf{n})^2 \\ &\text{s.t} \quad n_x^2 + n_y^2 + n_z^2 = 1 \end{aligned} \tag{4.11}$$

In the given function, p_i is an arbitrary point in the pointcloud defined in (2.45) containing N points. The centroid of the pointcloud is given by c, and the normal vector of the fitted plane is $\mathbf{n}$ The optimization problem was solved using the Python library SciPy which contained solvers for this type of problem. A program was written which was able solve the problem shown in (4.11), but was also able to utilize weights as this was the intention of the method in the next section. To solve the optimization problem, the SciPy function *minimize* was used which needed three different inputs in order to render a optimal solution. First, the cost function needed to be defined such that the minimize-function could evaluate the cost associated with the given iteration of the normal-vector. This was done by defining a function which corresponded to the cost-function (4.11), and is shown as *cost_func* on line 31 in appendix 10.

Next, the minimize-function needed to be provided an initial guess for the solution of $\mathbf{n}$. The code was tested using different values for the initial guess and it did not alter the solution of the problem. However, the normal vector outputted as the solution would change directions based on the initial guess. As they both spanned the same plane, this was not of concern. Moreover, depending on the initial guess, the convergence time would differ as the number of iterations would increase for initial guesses that were located far from the solution. For simplicity, the initial guess was set to an arbitrary vector on the unit-sphere.

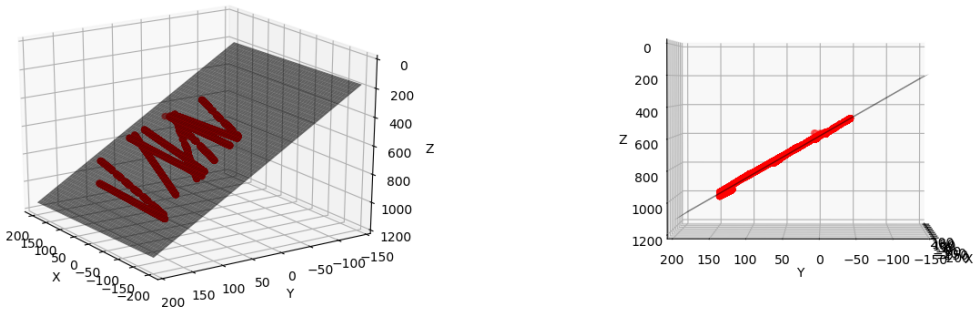
Lastly, the constraint of the vectors length had to be defined. Following the documentation of SciPy, the constraint was defined as a dictionary [37]. The dictionary contained information about what type of constraint was used (i.e equality-constraint), and a function which allowed the optimizer to calculate the value of the constraint. The function of the constraint can be seen on line 38 of appendix 10, and the dictionary describing the constrain is defined on line 40.

Furthermore, the method applied to the minimize-function was SLSQP – which was a suited method for solving a scalar cost-function with a nonlinear equality constraint [38]. As seen from the written code, the optimization solver had two inputs, *dictio* and *W*. This was mainly due to compatibility with weight

assignment and will be explained in detail in the next section. Furthermore, the output of the function was the point-normal representation of the laserplane. The estimated **n**-vector was the corresponding normal-vector and the centroid of the pointcloud was given as a point on the plane.

The finished function, called *opt_solver*, calculated the plane based on the pointcloud and the defined cost-function, which resulted in the following point-normal representation. The plot of the plane is illustrated with the pointcloud in Figure 4.15a-b.

$$\mathbf{n} \approx \begin{bmatrix} 0.01725 \\ 0.9335 \\ -0.3581 \end{bmatrix} \quad c = [19.95, 47.94, 714.65] \tag{4.12}$$



(a) Optimized plane

(b) Side view

Figure 4.15: Optimization plane representation

To supplement the solution given by the optimizer, another program was written that iterated through vectors on the unit-sphere and evaluated the cost-function. The program used spherical coordinates, and evaluated the cost corresponding to a total of 788768 different vectors on the unit-sphere.

The data was then illustrated using a Python package called PPTK (Point processing toolkit) [39] which allowed plotting of a large number of points. Figure 4.16 illustrates the unit-sphere where the constraint is active. The cost for the given vector is represented by the color of the point, where red represent high cost and blue is low cost. As seen from the figure, there are two high-cost regions and two low-cost regions. As explained previously the normal-vector tended to switch directions based on initial guess. The two solutions are represented by the center of the two low-cost areas illustrated in Figure 4.16.

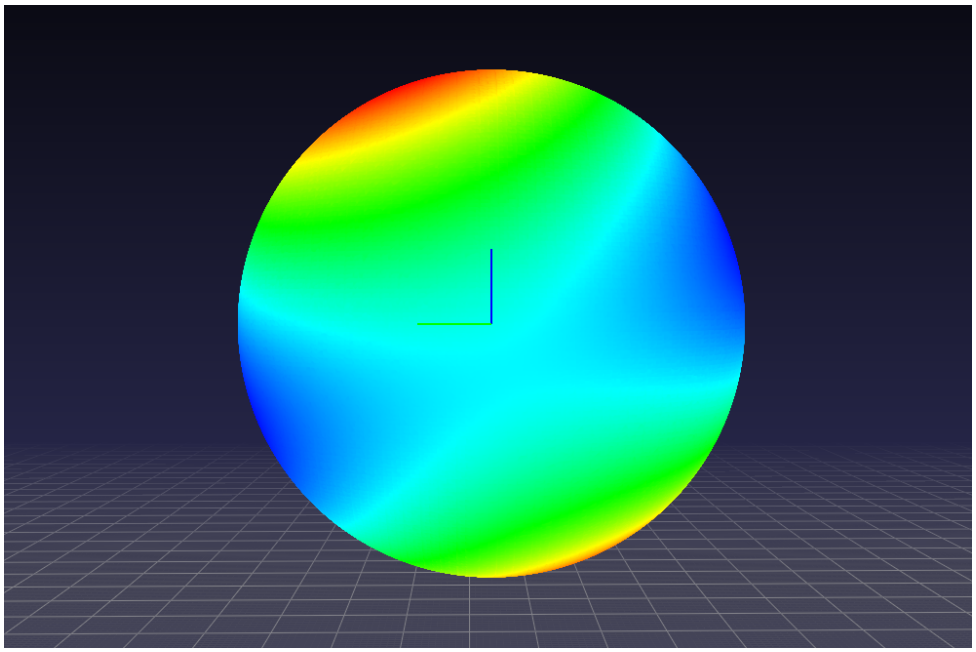


Figure 4.16: Cost illustrated on a unit-sphere

Furthermore, the normal-vector recovered from the optimizer was added to the sphere in order to identify the location of the sphere. Naturally, the point was located in the middle of low-cost area, marked with a red dot in Figure 4.17.

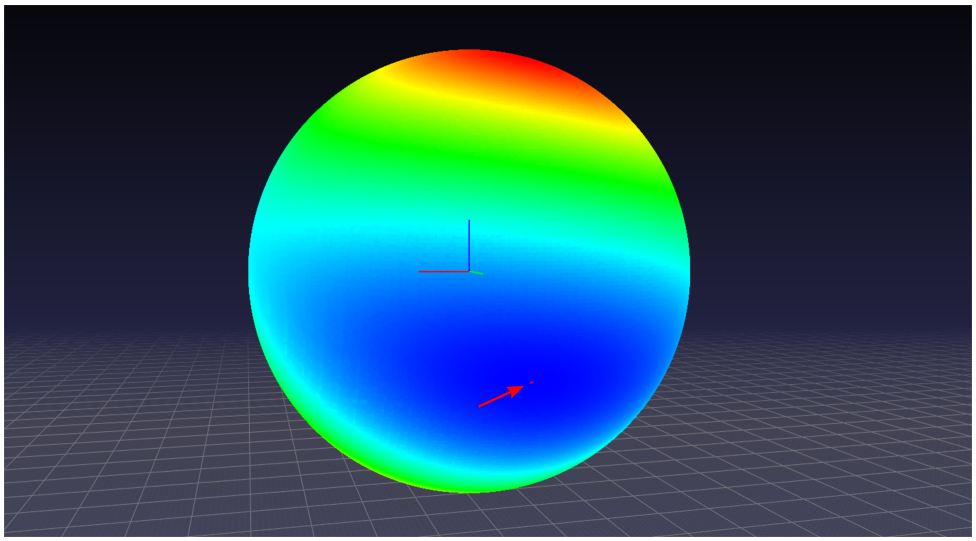


Figure 4.17: Marked solution from optimizer

4.2.3.3 Optimization with weights

As described in section 2.7.2, the rotational matrices and the translational vectors gathered from Matlab contained a certain amount of error. These errors translate to errors in the pointcloud, as both the rotation-matrix and translational-vector is directly used in the calculation of the points. As a result, the errors may cause the estimated plane to deviate from the actual laserplane. In an attempt to improve the estimation of the laserplane, this error was now to be taken into consideration.

Some of the images taken posed higher uncertainty than others. The goal was to force the estimated plane towards the points with less associated error in the pointcloud. This meant that the deviation from the more certain points (with less error) had to be penalized with a higher cost as well as it allowed to deviate from uncertain points. However, this meant that the points extracted from the pointcloud generation had to be labeled with which image it corresponded such that the correct weights could be assigned. Due to this, the previously mentioned code which generated the pointcloud (appendix 8) had to be modified. Instead of simply producing an array of spatial coordinates, it produced a dictionary which labeled the extracted points with the name of the corresponding .npy file. As the different .npy-files was through systematic naming directly linked to a corresponding image, the error to each extracted point could be accessed using their assigned label. This modification is shown in appendix 11.

Furthermore, a function was written which calculated the weights of the points based on the error of their corresponding checkerboard. Since both the errors in rotation and translation was given as vectors, the magnitude of error was calculated as the magnitude of the vector. The function used the following relationship in order to calculate the weights associated with each of the images.

$$W_j = \frac{1}{||\mathbf{e}_j^r|| + ||\mathbf{e}_j^t||} \tag{4.13}$$

From (4.13) it can be seen that if the error related to one of the checkerboards is large - the weight will tend towards a small number. Inversely, a small error will result in a larger weight.

The function, which is shown in appendix 12, used the error-vectors as input and calculated the weights of each checkerboard and saved the weights in a dictionary with their corresponding label. For the 10 images used in section 4.2.1, the dictionary had the following format, where W_j is the weight to the given checkerboard calculated by (4.13).

$$\mathbf{W}_j = \begin{bmatrix} \text{"1"} : W_1 \\ \text{"2"} : W_2 \\ \vdots \\ \text{"10"} : W_{10} \end{bmatrix}$$

Moreover, as the dictionary containing the labeled pointcloud was provided to the *opt_solver*-function, coordinates of the points are recalculated as the distance between each point and the centroid. Note that the number n is not necessarily the same of each of the labels as the different images had different amount of extracted points.

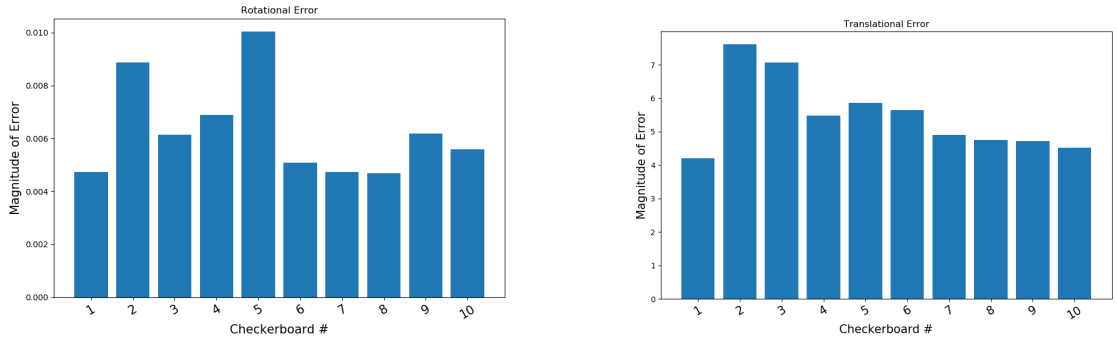
$$\mathbf{D}_j = \begin{bmatrix} \text{"1"} : p_1, ..., p_n \\ \text{"2"} : p_1, ..., p_n \\ \vdots \\ \text{"10"} : p_1, ..., p_n \end{bmatrix} \rightarrow \begin{bmatrix} \text{"1"} : p_1 - c, ..., p_n - c \\ \text{"2"} : p_1 - c, ..., p_n - c \\ \vdots \\ \text{"10"} : p_1 - c, ..., p_n - c \end{bmatrix} \tag{4.14}$$

Furthermore, a cost-function was added to the optimizer-function described in the previous section. The function was called *sum_func_W* and is shown in line 25 in appendix 10. The function iterated through the distances stated in $\mathbf{D}_j$, calculated the square of the distances and multiplied the previously defined cost with the corresponding weight.

$$\sum_{j=1}^{10} \sum_{i=1}^{N} \mathbf{W}_j [(p_{j,i} - c) \cdot \mathbf{n})]^2 \tag{4.15}$$

Where j corresponds to the label, and i is the iteration of the jth entry in $\mathbf{D}_j$ and $\mathbf{W}_j$.

From the 10 images of checkerboards used in generating the pointcloud, the error corresponding the rotation is shown in Figure 4.18a-b.



(a) Error in rotation

(b) Error in translation

Figure 4.18: Calculated error

Furthermore, based on the errors related to each image of the checkerboard – the following weights was calculated and applied to the optimizer.

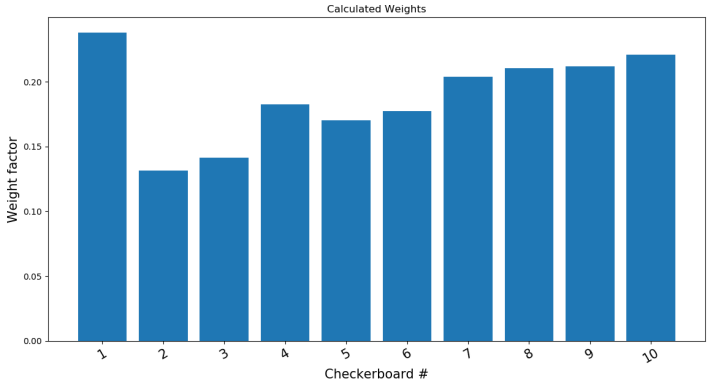
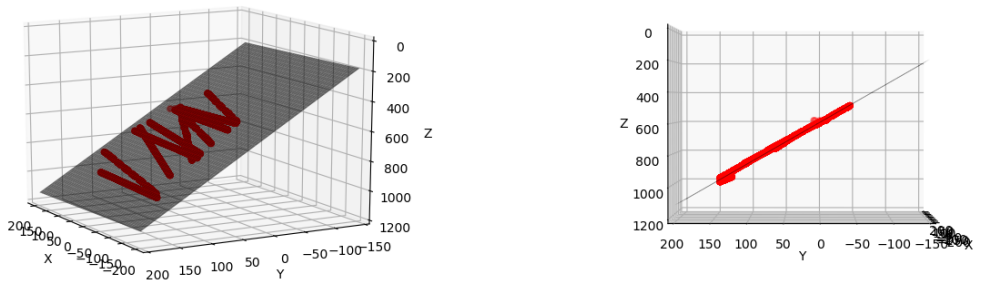


Figure 4.19: Calculated weights

Given the pointcloud and the weights shown in Figure 4.13, the optimizer calculated the following plane as the optimal solution. As the centroid of the pointcloud is calculate in the same manner, the c-component remained the same as the previously estimated plane.

$$\mathbf{n} \approx \begin{bmatrix} 0.01785 \\ 0.9327 \\ -0.3600 \end{bmatrix} \quad c = [19.95, 47.95, 714.65] \tag{4.16}$$



(a) Weighted plane

(b) Side view

Figure 4.20: Weighted optimization, plane representation

4.3 Estimated laser-planes

In order to compare the estimated planes numerically, all three of the estimated planes needed to be expressed in standard representation. As the two planes estimated in section 4.2.3.2 and 4.2.3.3 were expressed with point-normal notation, they were converted to the standard plane representation using the function *planeify*.

Written in standard form, the estimated planes were as following.

$$\text{Least Squares:} \quad 0.04915x + 2.60167y - z + 588.64971 = 0 \tag{4.17}$$

$$\text{Optimizer without weights:} \quad 0.04818x + 2.60709y - z + 588.70779 = 0 \tag{4.18}$$

$$\text{Optimizer with weights:} \quad 0.04801x + 2.59103y - z + 589.48137 = 0 \tag{4.19}$$

As shown in the three equations, there was only small differences separating the estimated planes. Furthermore, in order to compare the plane representations with the actual setup, the angle between the different laser-planes and camera was calculated. Using an online calculator [40], the angle between the laserplane and the cameras plane where $y = 0$, was calculated.
The calculated angles were given as being,

Least Squares:	21.05°
Optimizer without weights:	21.00°
Optimizer with weights:	21.12°

As the angle between the camera and the laser was previously measured with a goniometer to be approximately 20°, these values were in compliance with the actual setup. Furthermore, the angles were used to calculate the distance between the camera and laser. As the constant values in the equations 4.17-4.19 corresponded to the distance in the cameras z-axis to the intersection of the laserplane – simple geometry could be utilized. However, due to the similar values the average of both the angles and the constant values was used.

$$d = \sin(21.0566°) \cdot 588.9463\text{mm} = 211.6\text{mm}$$

Compared to the distance on the actual setup that was measured to be in the region of 20 cm, the distance of 21.16 cm was considered to be a realistic value.

4.3.1 Accuracy of the estimated laser-planes

As it was difficult to distinguish the most accurate plane from the different plane-equations, an experiment was carried out to determine the accuracy of the different laser-plane estimations.

In preparation of this experiment, the cameras focus was readjusted for a smaller field of view. This was to increase the accuracy of the readings, as described by the resolutions in section 3.3. The focus of the lens was manually set to a distance of approximately 500mm. Using a 10mm checkerboard and following the calibration procedure described in section 2.7, several images was captured and provided to the camera-calibration software. As the purpose of this calibration was only to identify the intrinsic properties of the camera, the extrinsic parameters (e.g rotational matrices and translational vectors) was not saved.

Moreover, as the adjustments done to the focus of the camera would not affect the relationship between the camera and the laser – the estimated parameters of the laserplane was unaltered.

The identified camera-matrix and distortion-coefficients was identified as being

$$K = \begin{bmatrix} 2997.5 & 0 & 1020.3 \\ 0 & 2999.2 & 583.34 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{D}_c = \begin{bmatrix} -0.0807 & 0.3690 & -0.0041 & 0.0065 \end{bmatrix} \tag{4.20}$$

The new camera-matrix and distortion coefficients were saved locally, such that the dedicated triangulation function could access them by loading the *loadCaliParam*-function. Furthermore, in order to decide which of the estimated laser-planes to use in the scanning-process, an object with known dimensions was 3D printed. The shape and relevant dimensions of the 3D-printed object is shown in Figure 4.22. The different planes were tested in its accuracy along two different directions of the laser, namely the x-direction and the z-direction according to Figure 4.21.

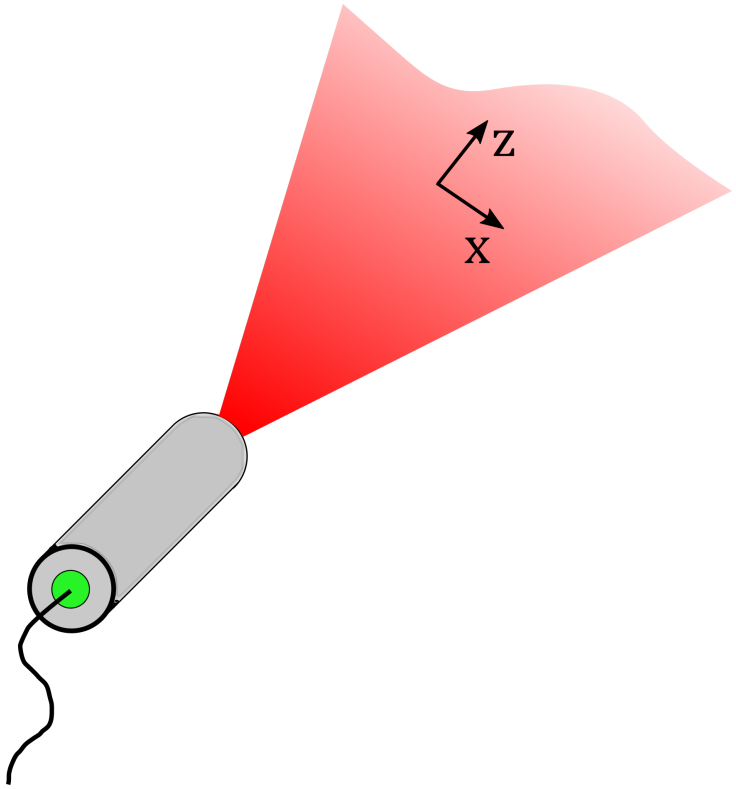


Figure 4.21: Directions of the laserline

Instead scanning the object once for each of the estimated laserplane, the object was scanned once and the pixel-coordinates was stored locally as a .npy file. This assured that the three different laser planes

were triangulated using the same pixels-coordinates as input, making the results more comparable.

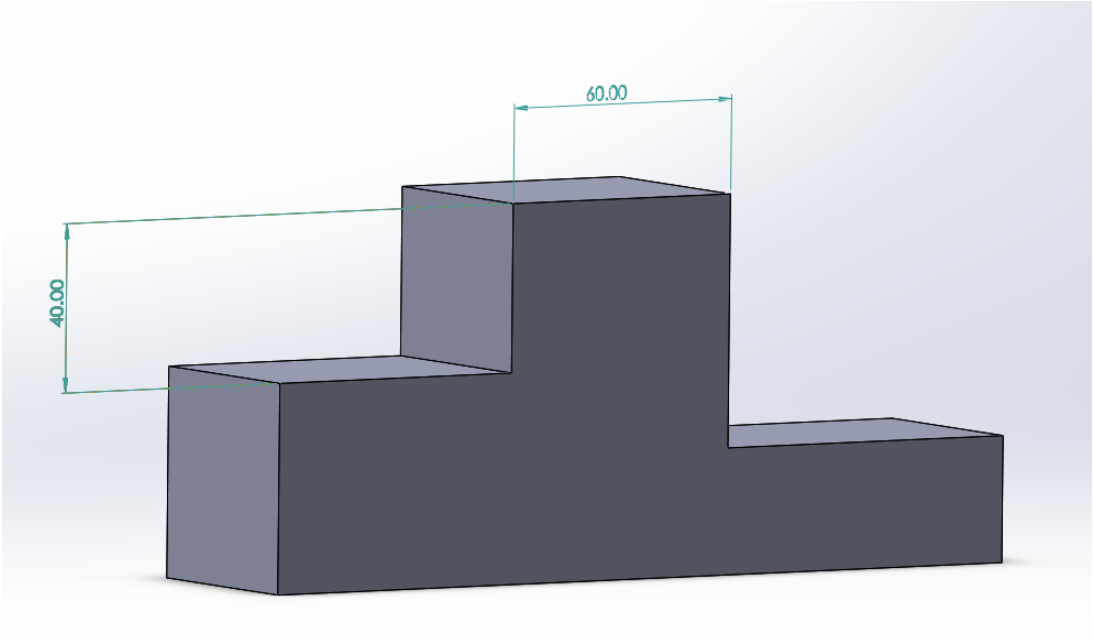


Figure 4.22: 3D-printed test object

4.3.2 Accuracy in x-direction

In order to determine the accuracy in the x-direction, one of the distances on the object was scanned as shown in Figure 4.23. The distance d, between the two points $\mathbf{P}_1$ and $\mathbf{P}_2$ was calculated using the following expression, where $\mathbf{P}_i$ is the spatial coordinate with respect to the camera frame.

$$d = \sqrt{(\mathbf{P}_{2,x} - \mathbf{P}_{1,x})^2 + (\mathbf{P}_{2,y} - \mathbf{P}_{1,y})^2 + (\mathbf{P}_{2,z} - \mathbf{P}_{1,z})^2} \tag{4.21}$$

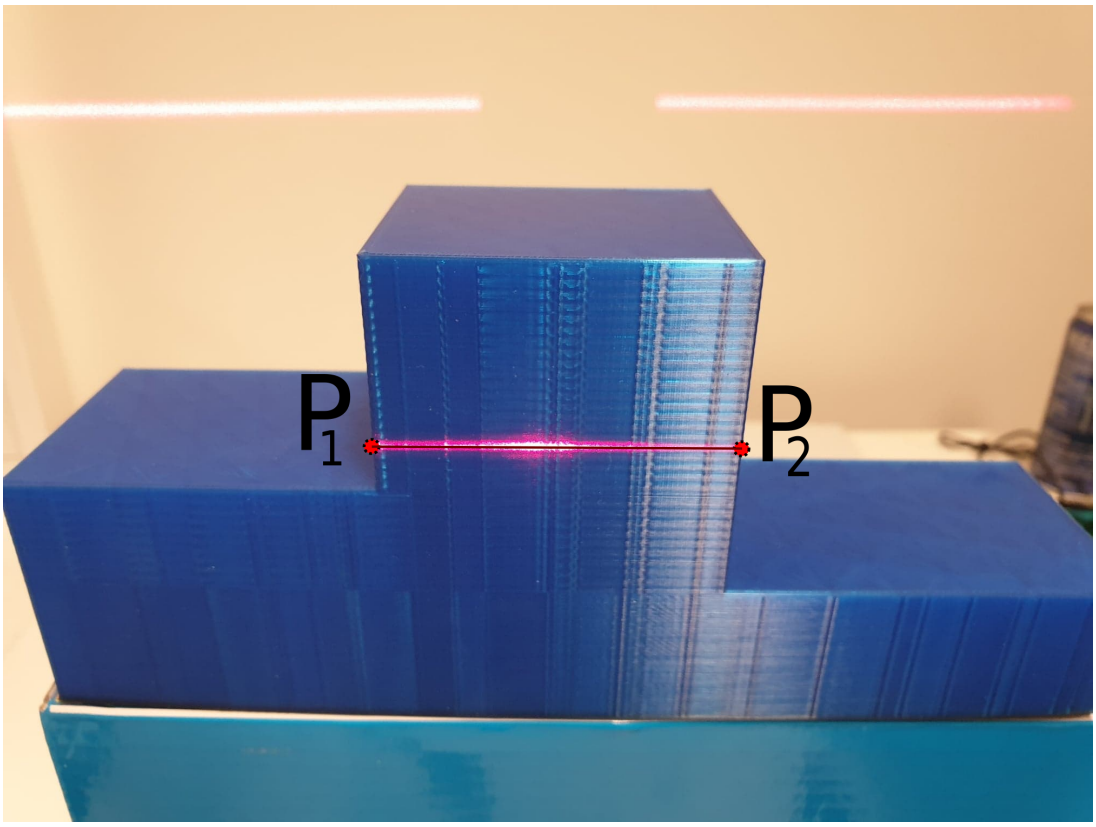


Figure 4.23: Laserline on 3D printed object

Based on the FOV and focal length of the camera, the estimated resolution in this direction is given by (3.7). The focal length of the camera can be estimated by analyzing the elements in the camera-matrix retrieved from the calibration, which is stated in (4.20). Furthermore, as seen in the definition of the camera-matrix in (2.33), the two elements 2997.5 and 2999.2 is the relationship between the focal length and pixel-size. By using this definition, the focal length can be estimated by the following calculation.

$$f \approx \frac{2997.5 + 2999.2}{2} \cdot 5.5\mu m \approx 16.49\text{mm} \tag{4.22}$$

The average of the two elements are used under the assumption that the pixels are square.

Moreover, as the focus of the camera was set to a distance of approximately 500 mm, the resolution could be calculated as

$$\Delta x = \frac{5.5\mu\text{m} \cdot 500\text{mm}}{16.49\text{mm}} \approx 0.1667\text{mm} \tag{4.23}$$

With the resulting theoretical resolution, the estimated laser-planes should be able to measure the distance from $\mathbf{P}_1$ and $\mathbf{P}_2$ with a deviation no larger than two times the theoretical resolution. This was the result of the resolution being applied to both points, which in a worst case scenario may cause the measured distances to deviate 0.3334 mm from the true value.

Furthermore, the object was scanned using the CoG-mode on the camera, which resulted in the pixel-coordinates shown in Figure 4.24. The laserline on the wall in the background of Figure 4.23 was not detected by the camera, which resulted in that the first and last pixel-coordinate in the array corresponded to $\mathbf{P}_1$ and $\mathbf{P}_2$, respectively.

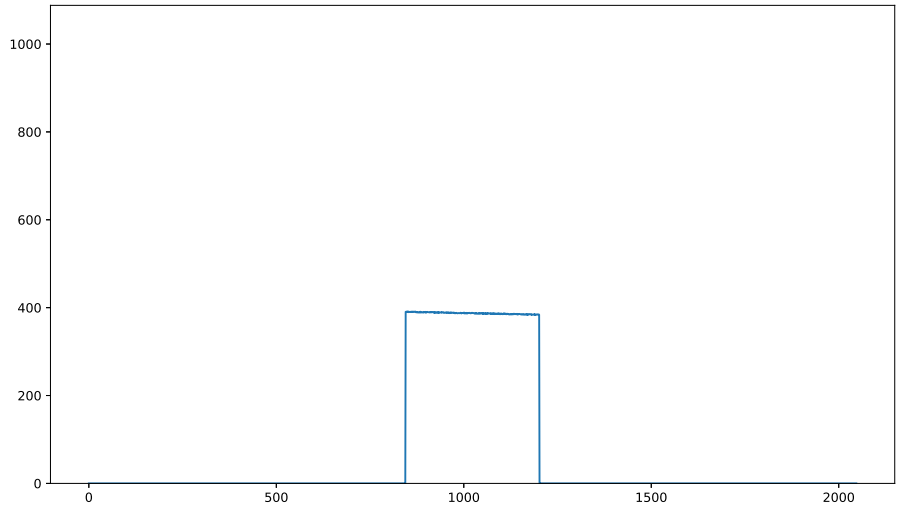


Figure 4.24: Extracted pixel coordinates, x-direction

The pixel-coordinates shown in Figure 4.24 was triangulated with the function described in section 3.6, with the three different planes that were estimated. From the design of the figure, the distance was expected to be 60 mm, which was confirmed with a caliper.

4.3.2.1 Least squares

As the different pixels were triangulated using the plane from section 4.2.3.1, the coordinates of $\mathbf{P}_1$ and $\mathbf{P}_2$ was calculated to be

$$\mathbf{P}_1 = \begin{bmatrix} -29.4678 & -32.3495 & 503.0382 \end{bmatrix} \quad \mathbf{P}_2 = \begin{bmatrix} 30.2890 & -33.4016 & 503.2381 \end{bmatrix}$$

The distance between the two points were then calculated by (4.21) to be approximately 59.76 mm.

4.3.2.2 Optimized plane

The plane which was defined in the triangulation function was changed to the one calculated by the optimizer without weights. Point $\mathbf{P}_1$ and $\mathbf{P}_2$ was given to be

$$\mathbf{P}_1 = \begin{bmatrix} -29.4633 & -32.3446 & 502.9625 \end{bmatrix} \quad \mathbf{P}_2 = \begin{bmatrix} 30.2812 & -33.3930 & 503.1078 \end{bmatrix}$$

These points resulted in a distance of 59.75 mm.

4.3.2.3 Optimized plane with weights

Lastly, the plane calculated with adding weights to the optimizer was used to triangulate the pixels. Using the same method as before, the resulting points was given to be

$$\mathbf{P}_1 = \begin{bmatrix} -29.5283 & -32.4160 & 504.0725 \end{bmatrix} \quad \mathbf{P}_2 = \begin{bmatrix} 30.3484 & -33.4671 & 504.2241 \end{bmatrix}$$

The distance between the two points where calculated to be 59.88 mm.

The following table could then be formed, where the deviation is the difference between the calculated distance and the measured distance of 60 mm.

Table 4.2: Accuracy x-direction

Plane	**Calculated distance**	**Deviation**
Least squares	59.76 mm	0.24 mm
Optimizer	59.75 mm	0.25 mm
Optimizer with weights	59.88 mm	0.12 mm

4.3.3 Accuracy in z-direction

In this part of the experiment, the object was translated on the table such that the pixels corresponding to both $\mathbf{P}_1$ and $\mathbf{P}_2$ of Figure 4.25 was present in the frame of the camera. As seen from Figure 4.22, the distance between $\mathbf{P}_1$ and $\mathbf{P}_2$ should be approximately 40 mm. However, when the distance was measured with a caliper, the actual distance was 40.5 mm. This was due to misprinting of the object, and might be the result of thermal expansion of the material. However, as this deviation was identified – this did not pose a challenge for the experiment.

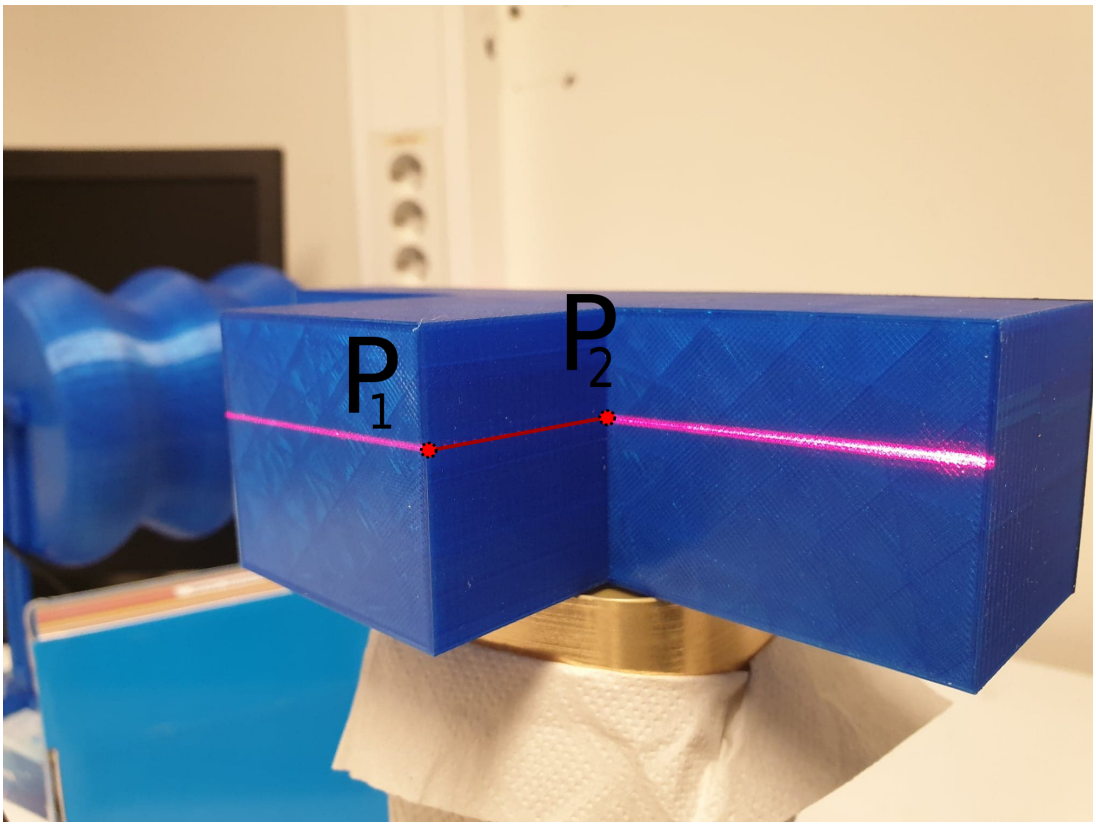


Figure 4.25: Laserline on object

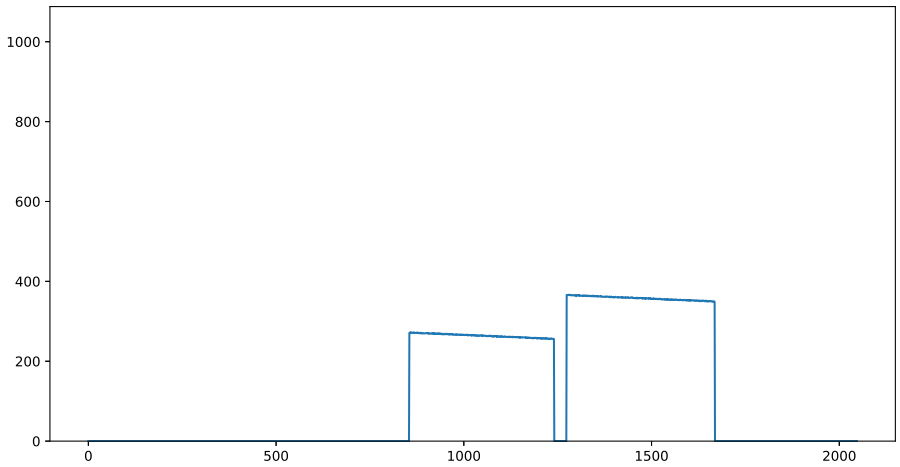


Figure 4.26: Extracted pixel coordinates, z-direction

As previously, the distance between the two points was calculated using equation 4.21. However, the points $\mathbf{P}_1$ and $\mathbf{P}_2$ could not be as easily extracted as previously. By analyzing the triangulated coordinates, the different triangulated points were identified by the sudden drop in its z-component.

As the resolution in the x-direction was calculated to be approximately 0.1667 mm, the resolution in the z-direction could be acquired by evaluating the following relationship.

$$\Delta z = \frac{\Delta x}{\sin \alpha} \tag{4.24}$$

The triangulation-angle α was previously found by analyzing the estimated planes, which averaged to approximately 21.0566°.

The resolution in the z-direction for the configuration could then be calculated as,

$$\Delta z = \frac{0.1667\text{mm}}{\sin\left(21.0566^\circ\right)} \approx 0.464\text{mm} \tag{4.25}$$

Similarly to the previous section, this value represents the smallest change in height that is detectable for the given configuration. The estimated laser-planes should ideally be able to determine the distance between $\mathbf{P}_1$ and $\mathbf{P}_2$ with a deviation no larger than 0.928 mm.

The pixel-coordinates shown in 4.26 was triangulated using the different plane-representations, and the points $\mathbf{P}_1$ and $\mathbf{P}_2$ were identified from the array of coordinates.

4.3.3.1 Least squares

By triangulating the points to the plane estimated by the least squares method, the following points were identified.

$$\mathbf{P}_1 = \begin{bmatrix} 33.6354 & -50.1306 & 459.8793 \end{bmatrix} \quad \mathbf{P}_2 = \begin{bmatrix} 42.0033 & -35.9719 & 497.1269 \end{bmatrix}$$

Applying these points to (4.21), the distance between the two points were calculated to be roughly 40.73 mm.

4.3.3.2 Optimized plane

Next, the pixel-coordinates was triangulated with the plane from the optimizer without weights. The points $\mathbf{P}_1$ and $\mathbf{P}_2$ were identified to be

$$\mathbf{P}_1 = \begin{bmatrix} 33.6213 & -50.1096 & 459.6869 \end{bmatrix} \quad \mathbf{P}_2 = \begin{bmatrix} 41.9906 & -35.9611 & 496.9768 \end{bmatrix}$$

Again, by using (4.21), the distance was calculated to be 40.74 mm.

4.3.3.3 Optimized plane with weights

Lastly, the plane resulting from the optimizer with the use of weights was

The points were identified as being,

$$\mathbf{P}_1 = \begin{bmatrix} 33.7113 & -50.2437 & 460.9169 \end{bmatrix} \quad \mathbf{P}_2 = \begin{bmatrix} 42.0866 & -36.0433 & 498.1128 \end{bmatrix}$$

The resulting distance between the points were calculated to be 40.68mm.

The following table was then formed, where the deviation is the difference between the calculated distance, and the measured distance of 40.5 mm.

Table 4.3: Accuracy z-direction

Plane	**Calculated distance**	**Deviation**
Least squares	40.73 mm	0.23 mm
Optimizer	40.74 mm	0.24 mm
Optimizer with weights	40.68 mm	0.18 mm

4.4 Rotary scan of cylindrical object

Due to time restrictions, it was not possible to travel to WellConnection such that an on-site experiment could be performed. Instead, the combined system were tested using a 3D-printed cylindrical shape, as shown in Figure 4.27. The object which was fitted with two holes, one to fit the shaft of the rotary encoder, and a hole on the opposite side for support. In addition, the object was designed with another set of holes 10 mm off the center in order to simulate non-centric rotation. In turn, the 10 mm offset from the center would imply that the distance between the object and laser would vary in the range of approximately 20 mm during one rotation. This was of interest as some of the drill-pipes at WellConnection's facility was reportedly rotating in a non-centric fashion.

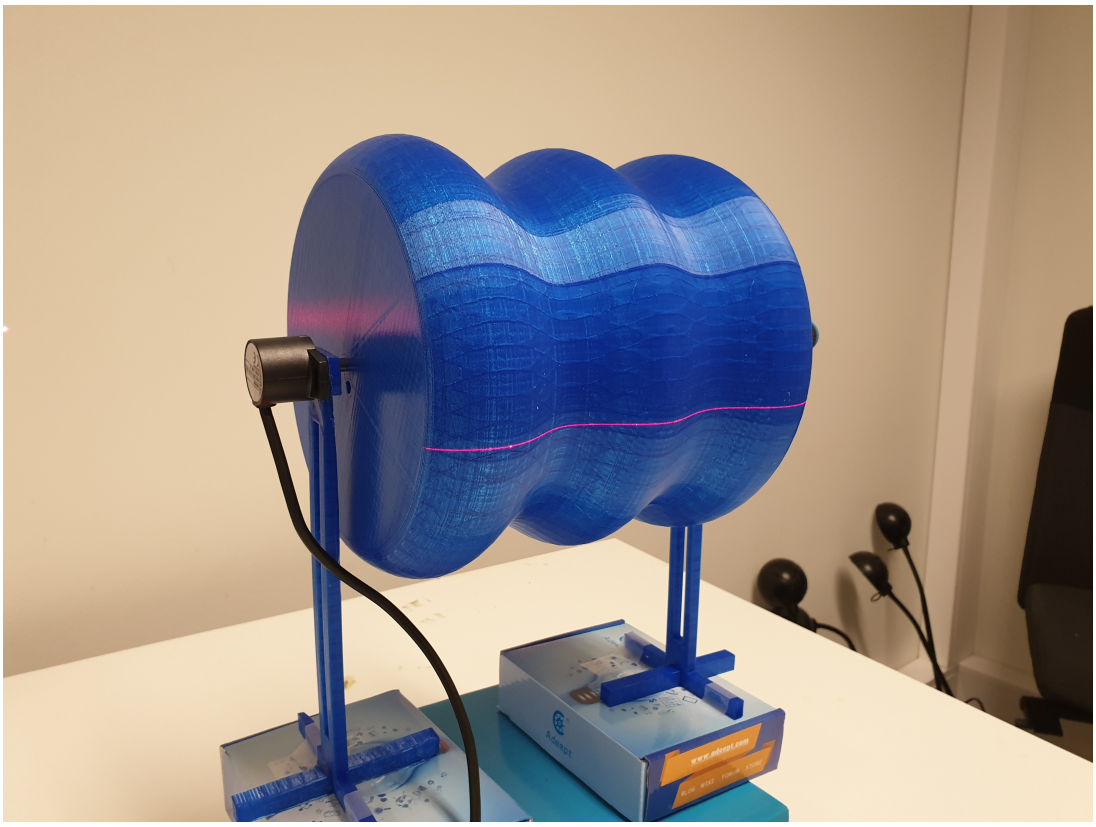


Figure 4.27: 3D-printed cylindrical object

As the program described in section 3.7 was initialized, the object was rotated one full rotation by hand. The rotation was done with caution, not to obstruct the path of the laserline. Furthermore, as the object had been rotated a full rotation, the scanning was terminated.

Figure 4.28 shows a sample of the generated dictionary, where this specific extraction corresponds to the angle 306.71°.

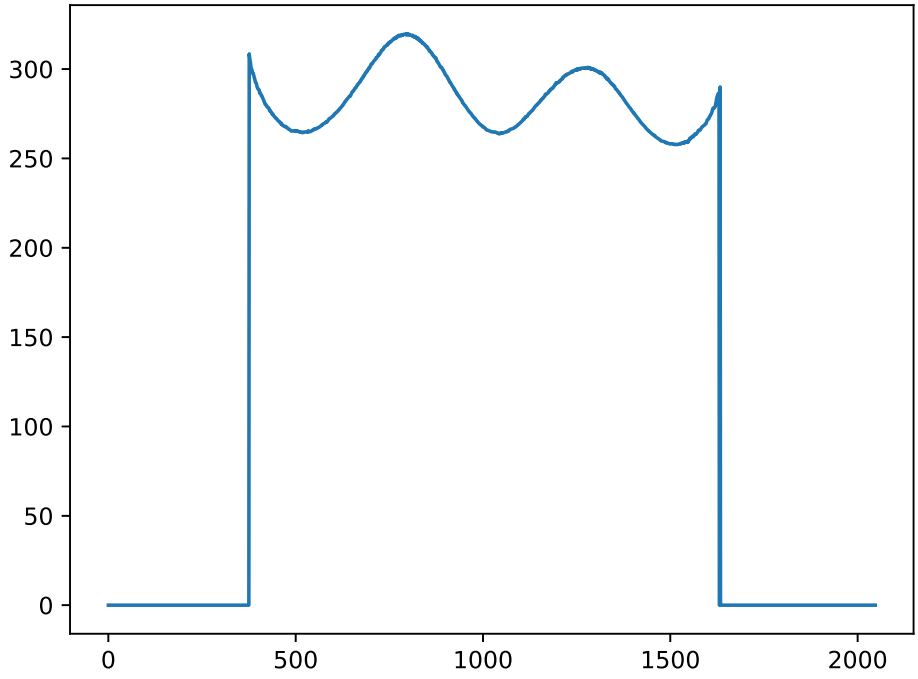


Figure 4.28: Sample of extracted pixel-coordinates at angle 306.71°

This process was then repeated with the support and encoder mounted in the off-center holes of the object which invoked a non-centric rotation. The resulting .npy-files containing both the angular readings and the pixel-extraction were saved locally such the data could be further processed.

However, upon inspection the files did not contain as many readings which was expected. The scan from the centric rotation contained pixel-extractions from 878 different angles, and the non-centric contained 1112. This was later identified to be caused by the cameras connection through USB, which heavily influenced the rate in which readings could be acquired from the camera. Moreover, as the laptop that was used in the experiments did not feature a connection port for the cameras communication cable, the readings were accepted as grounds for further processing.

4.5 Visualization of the data

In order to visualize the readings in a more apparent manner, the reading extracted in the previous section where transformed to an external coordinate system. In practice this would be a coordinate system which could be related to both the camera and the welding robot. However, in order to visualize the geometry which the readings represented – the homogeneous transformation to the rotary encoder was estimated. By capturing an image of a checkerboard approximately at the location of the encoder, the calibration-software of Matlab was once again utilized. The software returned the following rotation-matrix and translational vector corresponding to the checkerboard.

$$\mathbf{R}_{checker}^{cam} = \begin{bmatrix} 0.9993 & -0.0091 & 0.0359 \\ 0.0203 & 0.9454 & -0.3252 \\ -0.0310 & 0.3257 & 0.9450 \end{bmatrix} \quad \mathbf{t}_{checker}^{cam} = \begin{bmatrix} -63.13 \\ -69.66 \\ 541.81 \end{bmatrix} \tag{4.26}$$

From the theory regarding homogeneous transformation in section 2.3, the following transformation could be defined from the camera to the checkerboard.

$$\mathbf{T}_{checker}^{cam} = \begin{bmatrix} \mathbf{R}_{checker}^{cam} & \mathbf{t}_{checker}^{cam} \\ \mathbf{0} & 1 \end{bmatrix} = \begin{bmatrix} 0.9993 & -0.0091 & 0.0359 & -63.13 \\ 0.0203 & 0.9454 & -0.3252 & -69.66 \\ -0.0310 & 0.3257 & 0.9450 & 541.81 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.27}$$

However, the position of the encoder was not located at the origin of the checkerboard. The offset from the checkerboards origin to the encoder was approximately three squares in the checkerboards y-direction. As the checkerboard used had 10mm squares, this distance was roughly 30 mm. Furthermore, by assigning the encoders axis of rotation to be the x-axis, the rotation of the encoders coordinate system was included in the transformation. The angle θ would then represent the angular reading of the rotary encoder.

$$\mathbf{T}_{enc}^{checker} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 30 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.28}$$

The total transformation from the camera-frame to the encoder, was then given as

$$\mathbf{T}_{cam}^{enc} = \mathbf{T}_{checker}^{cam}\mathbf{T}_{enc}^{checker} \tag{4.29}$$

As the pixel-coordinates were being triangulated into spatial coordinates with respect to the cameras frame, they could be transformed into the encoders frame using (4.29).

$$\tilde{\mathbf{p}}^{enc} = \mathbf{T}_{enc}^{cam}\tilde{\mathbf{p}}^{cam} = \mathbf{T}_{cam}^{enc^{-1}}\tilde{\mathbf{p}}^{cam} \tag{4.30}$$

Moreover, as the pixel-coordinates got triangulated they were expressed as coordinates in the cameras frame. As the experiments in section 4.3.2 and 4.3.3 indicated that the plane estimated with weights had better accuracy, the pixel-coordinates from the scanning process were triangulated using this plane. Furthermore, by applying the transformation 4.30, each of the points could be expressed in the encoders frame. Additionally, since the key of the dictionaries entry was the rotation – the points where rotated around the frames x-axis. By doing this transformation for all the triangulated points, a pointcloud was generated which represented the shape of the object. The pointcloud was then illustrated by the Python-package PPTK, where each point was coloured according to its distance from the encoders x-axis.

4.5.1 Centric readings

As seen in the Figure 4.29, the system was able to give a satisfactory representation of the objects shape by applying the transformation (4.30). The difference in color relates to the distance from the frames x-axis to the point – where a red color indicated a longer distance. The color is in that sense a reading of the radius at a given point. Moreover, the figure illustrates the centric motion during the scan by an even color around the cross-section of the representation.

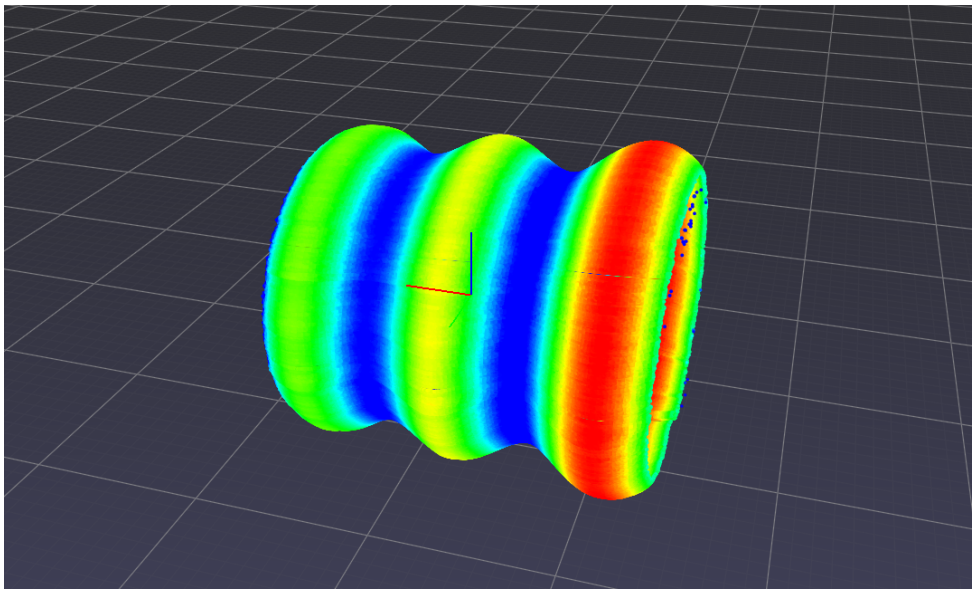


Figure 4.29: Centric representation - Sideview

Additionally, if the figure is seen from the front the centric motion becomes even more apparent. However, the figure demonstrates the consequence of the reduced number of readings, as some of the angles did not have a registered pixel-reading which is shown by the gaps around the figure.

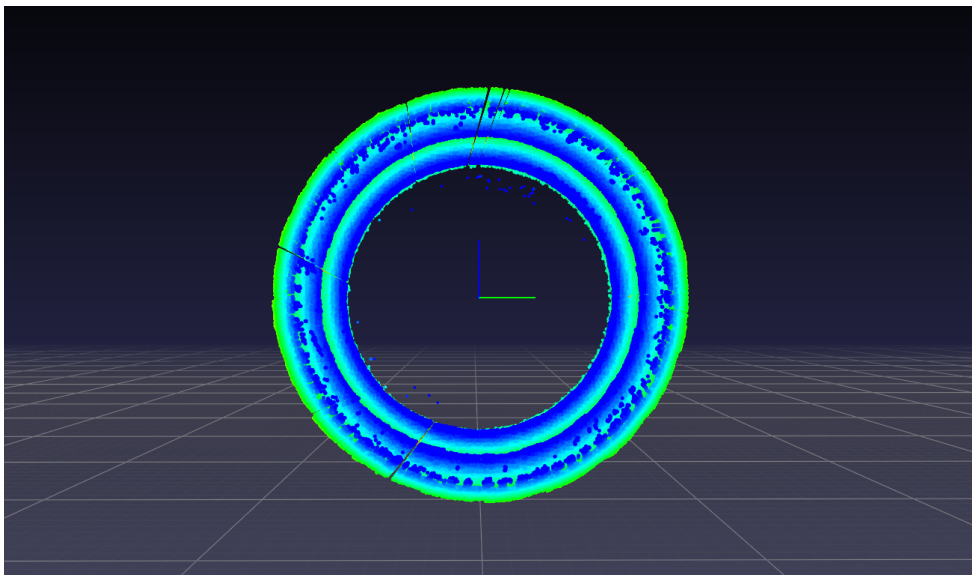
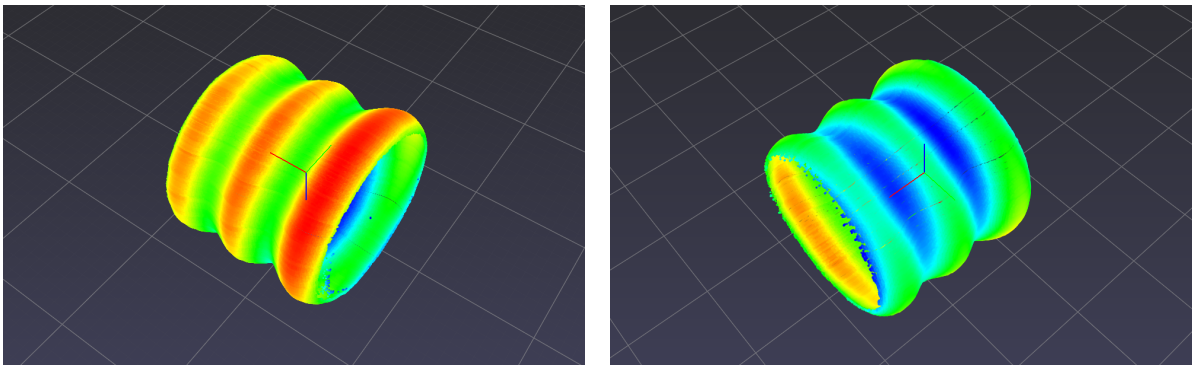


Figure 4.30: Centric representation - Frontview

4.5.2 Non-centric readings

The same transformation was applied to the readings acquired from the non-centric scan. Figure 4.31a-b illustrates the two sides of the figure which corresponds to the extremes of the distance between the object and the camera. Figure 4.31a corresponds to the readings where the object was closest to the camera, and the readings from 4.31b corresponds to the reading that was furthest away.

If the figure is seen from the front, the effects of the non-centric rotation is more easily noticed. Figure 4.32 demonstrates that as the object was rotated, the different readings where located at different distances from the camera.



(a) Extreme : Close (b) Extreme : Far

Figure 4.31: Non-centric representation

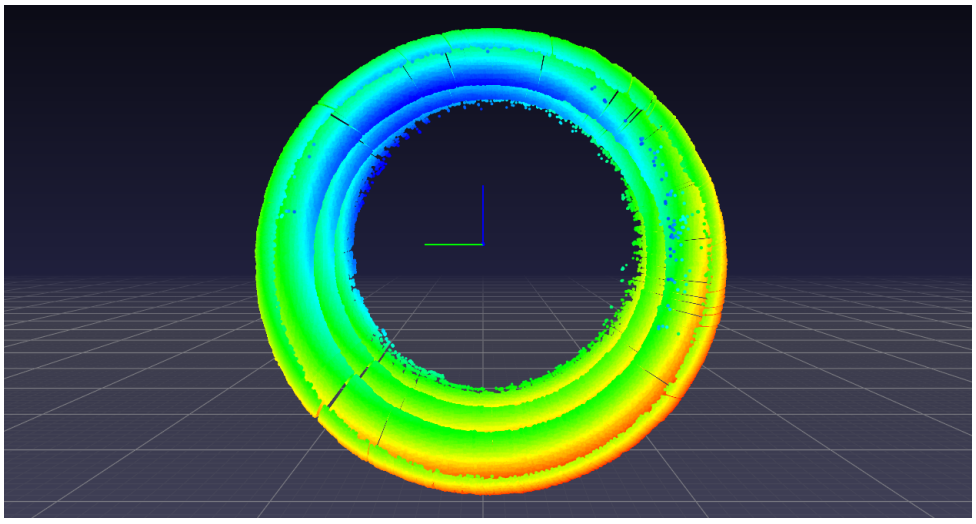


Figure 4.32: Non-centric representation - Frontview

4.5.3 Transformation to the laserplane

As the visualizations above does not directly contribute to information which can be utilized by a welding robot, a different homogeneous transformation is proposed. If the laserliner scans the object in an angle which represent a desired welding angle, the movement of robots end-effector can be locked in this plane. As a result, the coordinates can be expressed as 2D-coordinates in the laserplane consisting only of the x and z coordinates. As the coordinates corresponding to the objects curvature only occupies two dimensions, the third dimension can be used to relate the objects curvature to its orientation.

The desired transformation is illustrated in Figure 4.33, where the steps of the transformation is as follows.

1. $\mathbf{T}_1^c$: Translate the frame in the positive z-direction a distance of d_1
2. $\mathbf{T}_2^1$: Rotate the frame about the x-axis with the angle α (negative rotation)
3. $\mathbf{T}_L^2$: Translate the frame in the negative z-direction a distance of d_2

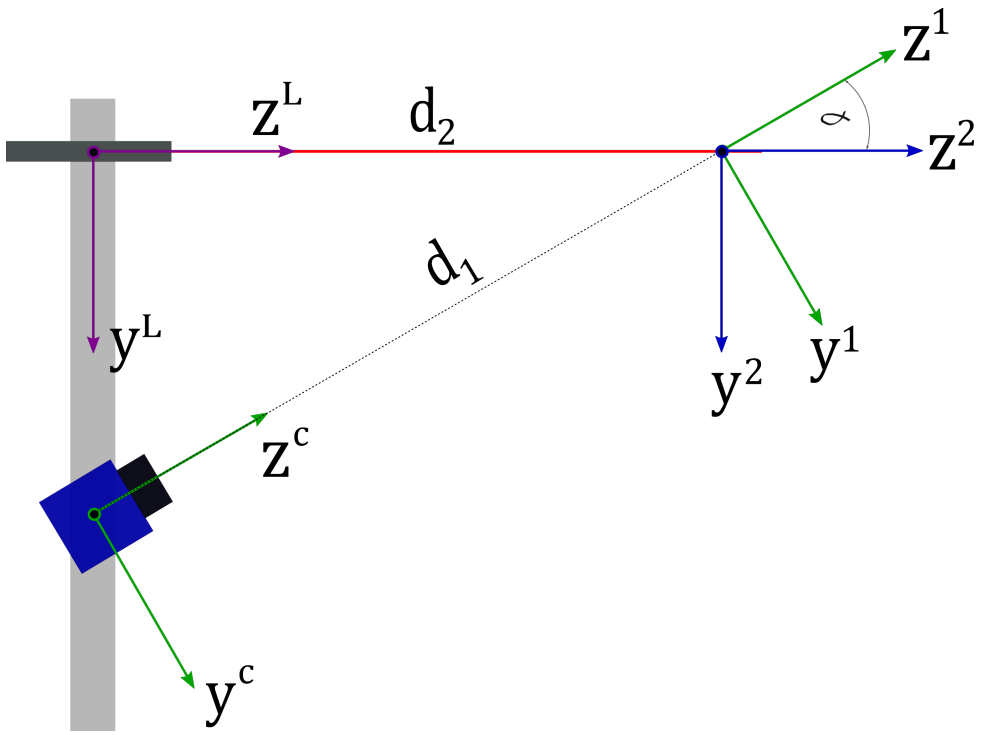


Figure 4.33: Transformation between camera and laser

In order to identify the dimensions d_1, d_2 and α – the information gathered from the analysis of the laserplane in section 4.3 was utilized. As the points were triangulated using the weighted optimized plane, the representation of this plane was used as basis in order to identify the dimensions. The distance d_1 corresponds to the constant in the representation in (4.19), which was 589.48mm. Furthermore, the rotation required to align the z-axis with the laserplane corresponds to the triangulation angle, which was calculated to be 21.12°. Lastly, the translation of a distance d_2 in the negative z direction can be computed with simple trigonometry where $d_2 = d_1 \cos(21.12°) \approx 549.88$mm.

Moreover, in order to implement the angular reading into the representation - the coordinates are translated in the y-direction of the laser-frame according to the angular reading.

Following the theory from homogeneous transformation in section 2.3, the required transformation were formed.

$$\mathbf{T}_1^c = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{T}_2^1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(-\alpha) & -\sin(-\alpha) & 0 \\ 0 & \sin(-\alpha) & \cos(-\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{T}_L^2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -d_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.31}$$

With the inserted values for d_1,d_2 and α, the transformation $\mathbf{T}_L^c$ was calculated by performing the following multiplication of the matrices.

$$\mathbf{T}_L^c = \mathbf{T}_1^c \mathbf{T}_2^1 \mathbf{T}_L^2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.9328 & 0.3603 & -198.1357 \\ 0 & -0.3603 & 0.9328 & 76.5337 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.32}$$

In order to transfer the coordinates defined in the frame of the camera to this plane, the inverse $\mathbf{T}_c^L$ was computed.

$$\mathbf{T}_L^p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.9328 & -0.3603 & 212.4033 \\ 0 & 0.3603 & 0.9328 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.33}$$

Such that,

$$\tilde{\mathbf{p}}^L = \mathbf{T}_c^L \tilde{\mathbf{p}}^c \tag{4.34}$$

On this form, a fourth transformation was applied. This transformation translated the points in the y-direction based on the angular measurement that they corresponded to.

$$\mathbf{T}_p^L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & \theta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.35}$$

The code which was written for the homogeneous transformation to the frame of the encoder and the laserplane is stated in appendix 17 and 18, respectively.

4.5.3.1 Centric scan

Applying the stated transformations to the readings recovered from the centric scan, the shape in Figure 4.34 is given. The distance from the laser-frame to the points are coloured depending on their z-value, where blue and red corresponds to shorter and longer distances, respectively. The coordinates are spanning the y-axis with their corresponding angle measurement θ, making the figure a function of curvature with respect to orientation.

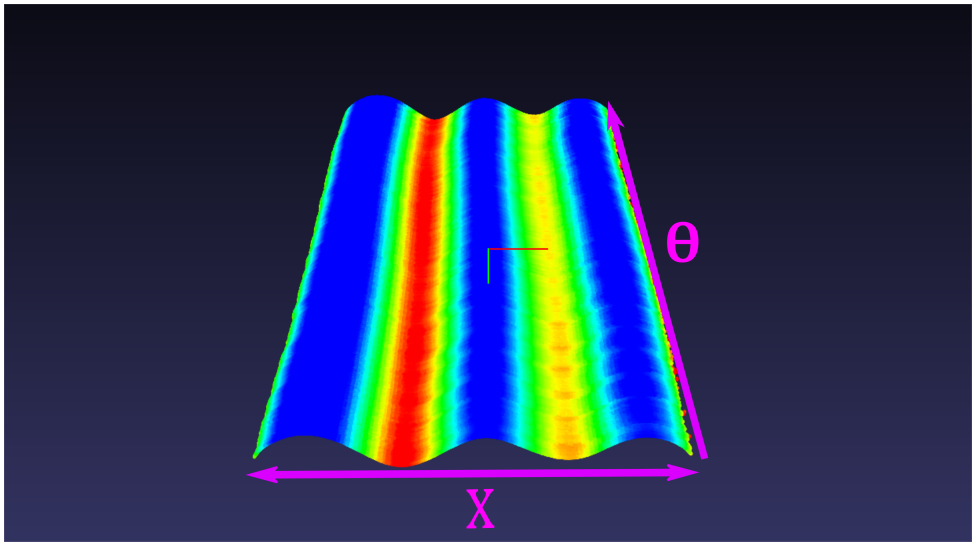


Figure 4.34: Centric : 2D data translated in angle dimension

Since the rotation of the object was centric while these readings where extracted, it can be seen that the distance from the laser to the different sections of the object does not vary much as the object is

rotated. Consequently, if the robot was to weld the leftmost groove, which corresponds to the red area, the robot could place its welding gun with the corresponding x-coordinate and z-value. Due to the lack of colour changes in the θ direction, the end-effector would not have to compensate for major changes in the z-direction.

4.5.3.2 Non-centric scan

The same transformation was done with the readings from the non-centric scan. As seen in Figure 4.35, the distance to the different sections of the object varies as it is rotated. This was however expected as the distance from the laser to the object changed due to off-center rotation. If a robot were in this case to weld the leftmost groove, it would have to compensate for the distance change in order to follow the curvature of the object.

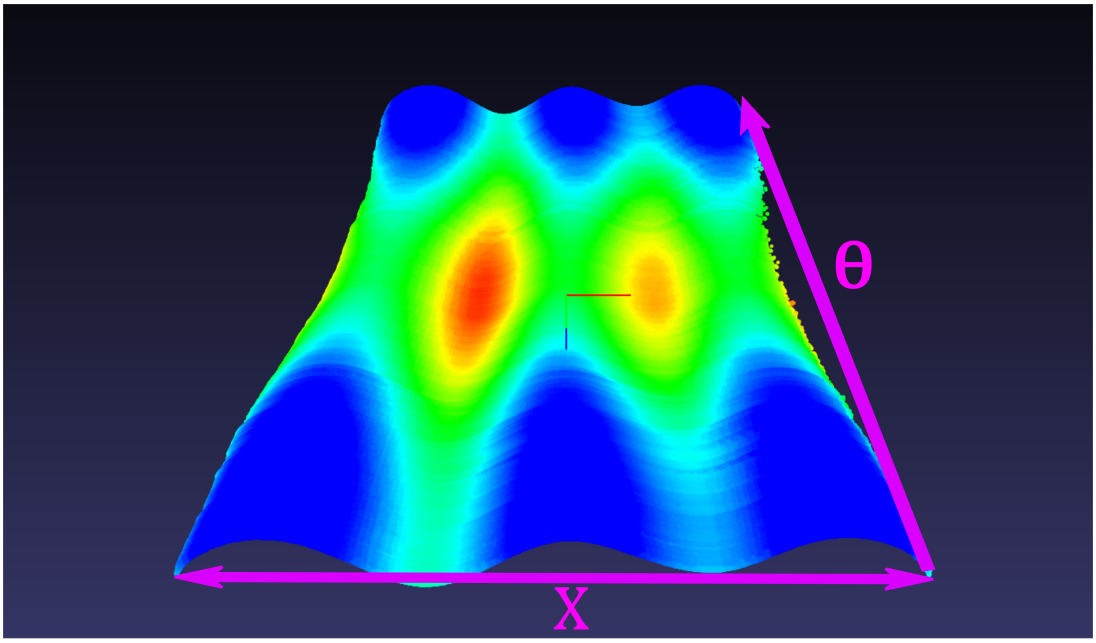


Figure 4.35: Non-centric : 2D data translated in angle dimension

4.6 Path estimation

Initially, the experiments and processing of data was concluded with the representations shown in the previous sections. As an additional experiment, the path along the leftmost groove of the non-centric readings is to be found. By inspection of the pointcloud shown in Figure 4.35, it was found that the x-coordinate corresponding to the leftmost groove was approximately -7.0901. In theory, by collecting all of the points with the same x-coordinate – the path formed in the theta direction should be extracted as demonstrated in Figure 4.36. However, as the intensity-readings with the CoG-method uses 6 sub pixels, it is highly unlikely that all the different readings along the θ-direction has a point which corresponds to this exact x-coordinate. In fact, after iterating through all the points in the pointcloud – no other points was found with precisely this x-coordinate.

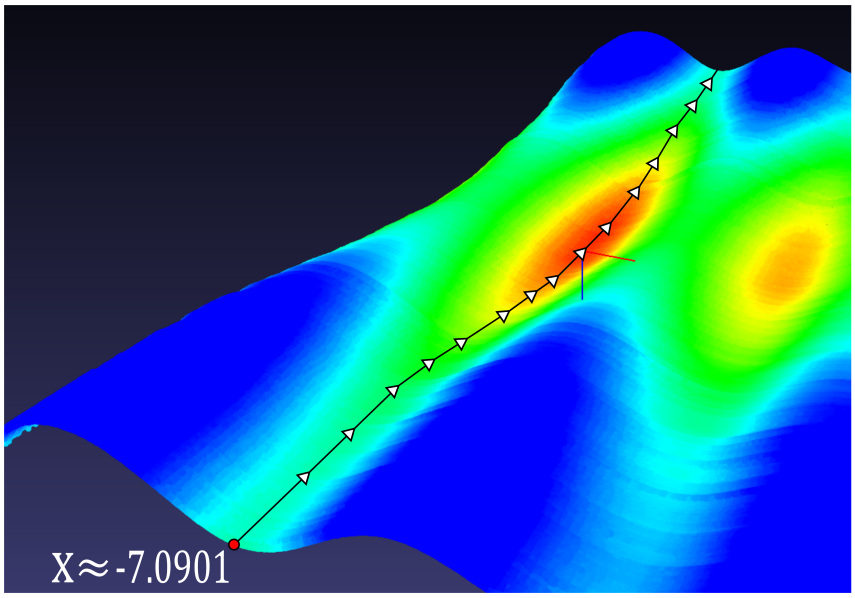


Figure 4.36: Desired path

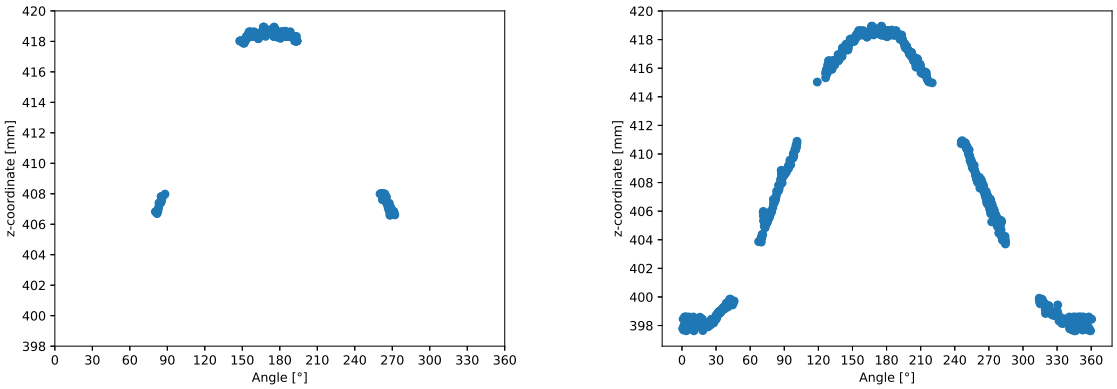
However, by implementing slack in the point-extraction, neighbouring points of the original x-coordinate can be included. Assuming the slack is sufficiently small, the additional points should result in inaccurate results. The neighbourhood of points was defined as

$$X_n \in [x - \epsilon, x + \epsilon] \tag{4.36}$$

First, the slack-parameter ϵ was set to be 0.01 mm, meaning that neighbourhood of points X_n was given by

$$X_n \in [-7.1001, -7.0801] \tag{4.37}$$

The points in this interval is shown in Figure 4.37a. As illustrated by the figure, the slack of 0.01 mm resulted in points being added to the path. However, due to the large spaces between the readings, ϵ was increased further. As the slack parameter was set to 0.05, additional points where introduced to the path. As seen from Figure 4.37b the spaces between the readings was reduced.



(a) $\epsilon = 0.01$ mm

(b) $\epsilon = 0.05$ mm

Figure 4.37: z-coordinate and angle, $\epsilon = 0.01$mm and 0.05mm

Lastly, ϵ was set to 0.075mm. As shown in Figure 4.38, the path from 0 to 360 degrees has sufficient z-coordinates from which a path can be recognized. Naturally, the top portion of the curve corresponds to the read area in Figure 4.36 where distance is the furthers from the previously defined laser-frame. Moreover, as it is not preferred to change the position of the end-effector in terms of discrete points – this data was further processed used to calculate a continuous curve.

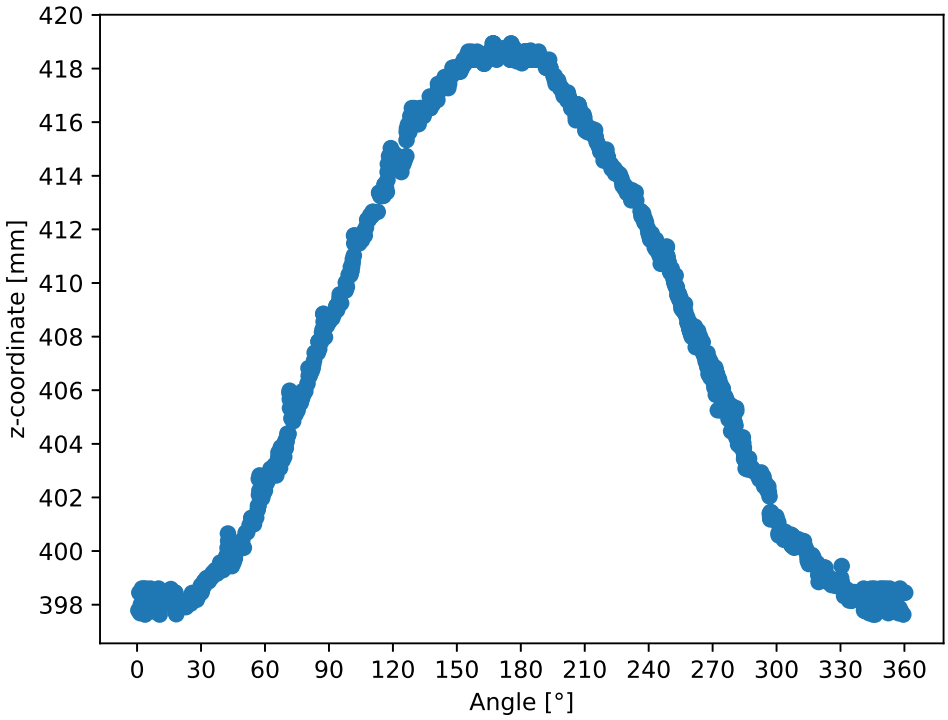


Figure 4.38: z-coordinate and angle, $\epsilon = 0.075$ mm

For demonstrative purposes the points were further treated using SciPy's function *polyfit* in order to retrieve a continuous curve representing the varying z-coordinate. Using the datapoints as input, the datapoints are fitted to a polynomial of the desired degree [41]. The curve is fitted according to the least squares problem

$$E = \sum_{j=0}^{k} |p(x_j) - y_j|^2 \tag{4.38}$$

After experimenting with different degrees of the function, the polynomial with a degree of 8 was found to be the most resembling to the data given. The average error between the curve and the data-points was stated to be approximately 0.081 mm. The estimated curve is illustrated in Figure 4.39, and the code is stated in appendix 16.

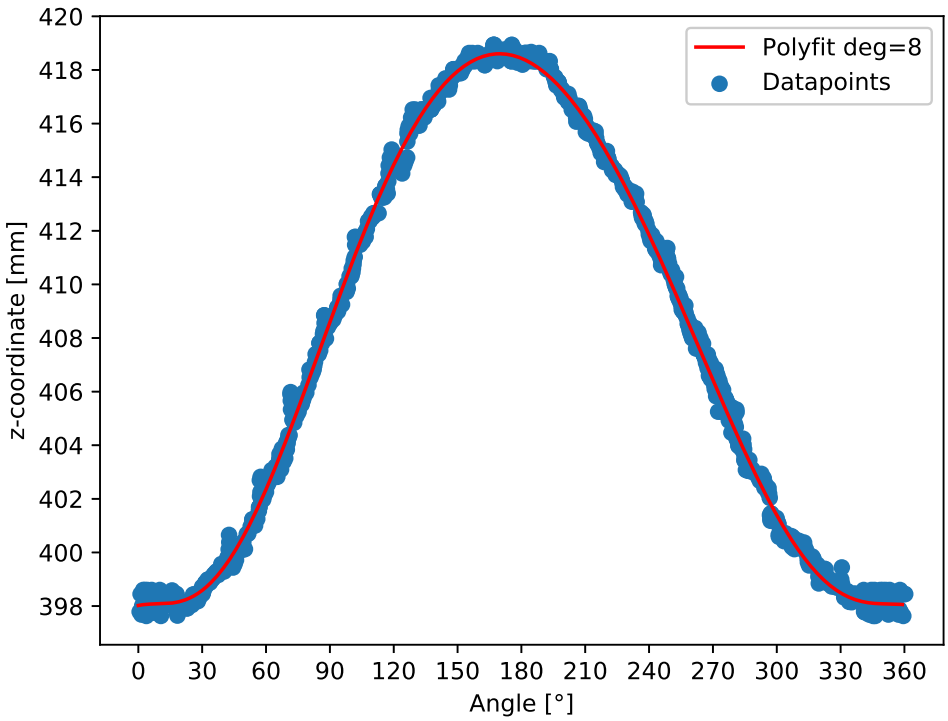


Figure 4.39: PolyFit processed path, $\epsilon = 0.075$ mm, deg $= 8$

5. Discussion

In this chapter both the results and the methods performed to acquire them are discussed. In this regard, some weaknesses related to the methods are mentioned, in which potential improvements are suggested. Additionally, limitations of the performed experiments are discussed since they were not performed with on-site conditions.

5.1 Laserplane calibration

From the laser/camera-configuration used in this thesis, the theoretical resolution is considered sufficient when compared to the recommended accuracy when providing an input to a MIG-welding robot. The results from the test performed on the three different laserplane-estimations indicated that their accuracy was within the expected value with the given configuration. However, the plane that was estimated with the use of weights proved to be more accurate when compared to the actual distances measured on the object. This indicates that there are benefits associated with considering the magnitude of error corresponding to the points in the pointcloud, compared to treating the points equally. However, the function which calculates the weights corresponding to the errors can be further improved. As the translational error tends to have a larger magnitude than the error of rotation, simply adding them is generally not the best solution. For a more accurate weight system, the error in rotation specifically with respect to the checkerboards x- and y-axis should be more heavily punished. This is reasoned from the fact that the error of these two axes directly influence the orientation of the plane which the pixel-coordinates of the laser are being triangulated with. In addition, when the pointcloud of the laser was being generated – there was a potential for disparities between the extracted pixel-coordinates and the position of the checkerboard. This was due to the image- and intensity-reading being asynchronous operations, and the checkerboard was held by hand. If the checkerboard did not remain stationary between the two operations, the extracted pixel-coordinates would not be correctly related to the image of the checkerboard. To minimize this effect, it is recommended to mount the checkerboard to a stand such that the checkerboard remains completely stationary between the two operations.

Furthermore, as the final frame was transformed to the plane of the laser, this was done under the assumption that the laser was perfectly aligned with the cameras horizontal direction. As seen from the equations of estimated planes, the coefficient corresponding the the x-variable is non-zero. This coefficient reflects the fact that a perfect alignment was not accomplished between the laser and camera. In turn, this translates to that the coordinates which was stated to be only in two dimensions, have a slight contribution in the third dimension. Since the third dimension was dedicated for the orientation measurement, this contribution should be minimized. As it can prove to be difficult to perfectly align the laser and camera, the simpler solution will be to add an additional rotation to the transformation corresponding to the misalignment.

5.2 Path estimation

As seen in the estimated path from the results, the path demonstrates an accurate compensation for the non-centric rotation. As the object was scanned during the non-centric rotation, the distance between

the laser and object would vary 20 mm according to the 10 mm offset. This distance is reflected by the estimated path of the z-coordinate, as it varies from roughly 398 mm to 418.5 mm for a total of 20.5 mm. As the value indicates, there is a deviation of 0.5 mm from the expected value of 20 mm. This deviation can be justified by considering the theoretical resolution in the z-direction. As the resolution was calculated to be approximately 0.464 mm, this could potentially result in a deviation of 0.928 mm between the far and close side of the object. In addition, misprinting of the object might also cause some deviation, but this was not considered as the deviation was within the theoretical resolution.

5.3 Limitations of the experiments

Since the results of the experiments are based on staged conditions, there are some aspects which needs to be addressed. As the intended use of the system is to extract readings from machined drill-pipes, there are challenges in terms of reflections which are not covered in this thesis. Since the surface of the machined metal is to be considered reflective, the intensity of the reflected laser might cause pixels in the camera to saturate. This effect may introduce inaccurate pixel-readings from the camera, which in turn will result in inaccurate reading of the curvature. The implications of reflections may be controlled with the use of a multiple slope-function, but since the object used in the experiments did not have a surface comparable to metal, the effect of this was not documented. Additionally, during the experiments the orientation of the object was acquired by simply connecting the encoder into a fitted socket. For an on-site application, this manner of mounting the encoder is not as feasible. A potential solution would be to mount the encoder to a surface that rotates in the same fashion as the pipes, such as the rotary chuck that drives the pipes rotation. In that sense, a direct measurement from the axis of rotation is recommended, as it does not rely on knowing the exact dimension of the pipe.

5.4 Compliance with the optimal welding position

It was stated by WellConnection that the optimal weld was achieved by placing the welding gun on the top portion of the drill-pipe. To accommodate this, the top section of the pipe can be scanned such that the laserline corresponds to the desired position of the welding gun. Consequently, the extracted readings will correspond to the desired location of the welding gun as the drill-pipe is rotated. Furthermore, if the end-effector is locked in the plane which was spanned by the laser during scanning, the only compensation of movement required are the distance given by the varying height-function.

6. Conclusion and further work

6.1 Conclusion

This thesis demonstrates how the combination of angular measurement and laser triangulation can be utilized to provide a positional reference for a potential welding robot. The calculated path of the welding robot demonstrated a clear correspondence to the 20mm distance variation caused by the non-centric rotation. Additionally, the theoretical resolution of the laser/camera-configuration is regarded sufficient to be utilized as input for a MIG-welding robot. The most accurate estimation of the laserplane was found by considering the individual errors corresponding to the points of the pointcloud. When this plane was used to measure distances in the x- and z-direction, the deviation from the actual distances were 0.12mm and 0.18mm, respectively. However, due to the nature of the experiments, there is an underlying uncertainty related to the performance of the laser triangulation on the metal surface of the drill-pipes. As this effect was not covered in the experiments, the author is unable to conclude with an absolute conclusion in regards to feasibility. If the use of functions such as multiple-slope can ensure a reliable extraction of pixel-coordinates despite the reflection, the proposed method should be a suitable method of supplying a positional reference to a welding robot in terms of accuracy.

6.2 Further work

In order to determine if the proposed system can be utilized for machined drill-pipes, a study has to be performed in order to identify how the the pixel-coordinates are extracted from the reflective surface on machined metal. If the reflections result in the camera being unable to provide accurate pixel-coordinates, a different method should be considered.

Additionally, throughout the thesis it has been assumed that the frame of the camera could be related to the frame of the robot. This relationship needs to be identified if a robot is to utilize the points extracted from the camera. In practice, this is achieved by performing hand-eye calibration, but as explicit use of a robot was not part of this thesis – this type of calibration was not covered.

Lastly, the data presented in this thesis needs to be further processed. In this regard, a robust method which can compute the path of the robot needs to be derived. In addition, logic needs to be implemented such that the robot recognizes where to perform the weld.

Bibliography

[1] Movimed, "What is laser triangulation?" [Online]. Available: http://www.movimed.com/knowledgebase/what-is-laser-triangulation/

[2] D. Ramel, "Popularity index: Python is 2018 'language of the year'." [Online]. Available: https://adtmag.com/articles/2019/01/08/tiobe-jan-2019.aspx

[3] O. Egeland, "A note on robot kinematics," Handout.

[4] P. Sanz, "Robotics: Modeling, planning, and control (siciliano, b. et al; 2009) [on the shelf]," *IEEE Robotics Automation Magazine*, vol. 16, no. 4, pp. 101–101, December 2009.

[5] E. Liavik, "Utilizing laser triangulation to extract spatial coordinates of drill-pipe's geometry," Specialization Project at IKT/NTNU, Not publicly available, 2018.

[6] P. Corke, *Robotics, Vision and Control*. Springer, 2013.

[7] O. Egeland, "A note on vision," Handout.

[8] R. Sagawa and Y. Yagi, "Accurate calibration of intrinsic camera parameters by observing parallel light pairs," 06 2008, pp. 1390 – 1397.

[9] T. Hong, M. Ghobakhloo, and W. Khaksar, "6.04 - robotic welding technology," in *Comprehensive Materials Processing*, S. Hashmi, G. F. Batalha, C. J. V. Tyne, and B. Yilbas, Eds. Oxford: Elsevier, 2014, pp. 77 – 99. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B978008096532100604X

[10] W. Karwowski, M. Rahimi, H. Parsaei, B. R. Amarnath, and N. Pongpatanasuegsa, "The effect of simulated accident on worker safety behavior around industrial robots," *International Journal of Industrial Ergonomics*, vol. 7, no. 3, pp. 229 – 239, 1991. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0169814191900068

[11] K. Koskinen, P. Kohola, J. Hirvonen, J. Annanpalo, H. Lehtinen, and K. Vartiainen, "Experimental vision based robot system for arc welding," *IFAC Proceedings Volumes*, vol. 19, no. 9, pp. 123 – 130, 1986, 1st IFAC Workshop on Digital Image Processing in Industrial Applications, Espoo, Finland, 10-12 June 1986. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1474667017575199

[12] K. Weman, "12 - other methods of welding," in *Welding Processes Handbook (Second Edition)*, second edition ed., ser. Woodhead Publishing Series in Welding and Other Joining Technologies, K. Weman, Ed. Woodhead Publishing, 2012, pp. 133 – 142. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780857095107500124

[13] "Executive summary world robotics 2018 industrial robots'," International Federation of Robotics. [Online]. Available: https://ifr.org/downloads/press2018/Executive_Summary_WR_2018_Industrial_Robots.pdf

[14] E. Wai Yan So, M. Munaro, S. Michieletto, S. Tonello, and E. Menegatti, "3dcomplete: Efficient completeness inspection using a 2.5d color scanner," *Computers in Industry*, vol. 64, p. 1237–1252, 12 2013.

[15] R. Szeliski, *Computer Vision*. Springer, 2014.

[16] Matlab, "Computer vision toolbox." [Online]. Available: https://se.mathworks.com/help/vision/

[17] ——, "Single camera calibrator app." [Online]. Available: https://se.mathworks.com/help/vision/ug/single-camera-calibrator-app.html

[18] H. Alzarok, S. Fletcher, and A. P. Longstaff, "A new strategy for improving vision based tracking accuracy based on utilization of camera calibration information," in *2016 22nd International Conference on Automation and Computing (ICAC)*, Sep. 2016, pp. 278–283.

[19] L. Vandenberghe, "Solution of a least squares problem." [Online]. Available: http://www.seas.ucla.edu/~vandenbe/133A/lectures/ls.pdf

[20] X.-D. Zhang, *Matrix Analysis and Applications*. Cambridge University Press, 2017.

[21] *C4 2040 Hardware Reference Manual*, Automation Technology, 2015. [Online]. Available: https://www.stemmer-imaging.com/media/uploads/cameras/at/12/122200-Automation-Technology-User-Manual-C4-2040.pdf

[22] "Application note - the fir filter," Automation Technology, 2014. [Online]. Available: https://www.stemmer-imaging.co.uk/media/uploads/cameras/12/122195-Automation_Technology_AppNote_FIR_Filter.pdf

[23] D. Acharya, A. Rani, S. Agarwal, and V. Singh, "Application of adaptive savitzky–golay filter for eeg signal processing," *Perspectives in Science*, vol. 8, pp. 677 – 679, 2016, recent Trends in Engineering and Material Sciences. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2213020916301951

[24] *Z-LASER*, Stemmer Imaging. [Online]. Available: https://www.stemmer-imaging.com/en-ie/products/z-laser-z25m18s3-f-640-lp45/

[25] W. Latimer, "Understanding laser-based 3d triangulation methods," https://www.vision-systems.com/factory/article/16738248/understanding-laserbased-3d-triangulation-methods, 2015.

[26] *Machine Vision Lenses*, Fujinon. [Online]. Available: https://www.fujifilm.com/products/optical_devices/cctv/pdf/fujinon_machine_vision_lens_catalog.pdf

[27] A. Technology, "Cx calibration." [Online]. Available: https://www.stemmer-imaging.com/media/uploads/cameras/12/122192-Automation_Technology_AppNote_CX_Calibration.pdf

[28] L. He, S. Wu, and C. Wu, "Robust laser stripe extraction for three-dimensional reconstruction based on a cross-structured light sensor," *Appl. Opt.*, vol. 56, no. 4, pp. 823–832, Feb 2017. [Online]. Available: http://ao.osa.org/abstract.cfm?URI=ao-56-4-823

[29] O. A. Olsen, *Industrielle målemetoder*, 2nd ed. Jubok, 2015.

[30] "Individual-standard magnetic sensor," Contelec. [Online]. Available: http://www.contelec.de/uploads/media/Vert-X_06.pdf

[31] Contelec, "Vert-x 28 datasheet." [Online]. Available: http://www.contelec.ch/fileadmin/user_upload/contelec/Downloads/Datenblaetter/Englisch/Vert-X/Vert-X%2028/Vert-X_28_5V_10...90_Ub_e.pdf

[32] Ardunino, "What is arduino?" [Online]. Available: https://www.arduino.cc/en/guide/introduction

[33] C. Liechti, "Pyserial." [Online]. Available: https://pythonhosted.org/pyserial/

[34] Robin2, "Arduino comms using python." [Online]. Available: http://forum.arduino.cc/index.php?topic=598209

[35] Ardunino, "Analog read resolution." [Online]. Available: https://www.arduino.cc/reference/en/language/functions/zero-due-mkr-family/analogreadresolution

[36] A. Bill Earl, "Adafruit 4-channel adc breakouts." [Online]. Available: https://cdn-learn.adafruit.com/downloads/pdf/adafruit-4-channel-adc-breakouts.pdf

[37] SciPy.org, "Scipy.optimize.minimize." [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html#scipy.optimize.minimize

[38] J. Ypma, "Sequential quadratic programming (sqp)." [Online]. Available: https://www.rdocumentation.org/packages/nloptr/versions/1.2.1/topics/slsqp

[39] H. Technologies, "Point processing toolkit." [Online]. Available: https://github.com/heremaps/pptk

[40] "Online calculator. angle between two planes," OnlineMSchool. [Online]. Available: https://onlinemschool.com/math/assistance/cartesian_coordinate/plane_angl/

[41] "Polyfit." SciPy. [Online]. Available: https://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html

Appendices

```python
import numpy as np
import os
import glob

def loadCaliParam():
    #path to the folder where the parameters are saved
    caliParam_folder = "C:/Users/espen/OneDrive/Master/Laserplankalibrering/Bilder_og_koordinater/Matlab" # home pc
    os.chdir(caliParam_folder)

    #Mean Reprojection Error
    ret = np.loadtxt('MeanReprojectionError.txt')

    #All Reprojection Errors
    rets = np.loadtxt('ReprojectionErrors.txt')

    #The Intrisinc Matrix
    mtx = np.loadtxt('./CalibrationConstants/calibratedCameraMatrix.txt')

    #Rotation Matrices and translation vectors between the scene and the camera
    tvecs = np.loadtxt('TranslationVectors.txt')
    rMats = np.loadtxt('RotationMatrices.txt') # Note: this file contains all the scene/camera rotationmatrices for each picture. It needs to be reshaped from (#,3) into (#/3,3,3)
    shp = rMats.shape
    C = int(shp[0]/3)
    rMats = rMats.reshape(C,3,3)
    #Radial and tangential distortion coeffecients, dist = [k_1,k_2,p_1,p_2[,k_3[,k_4,k_5,k_6]]]
    dist = []
    rDist = np.loadtxt('./CalibrationConstants/calibratedRaddist.txt') #k_1 and k_2, => k_3 = 0, this leads to dist = [k_1,k_2,p_1,p_2]
    tDist = np.loadtxt('./CalibrationConstants/calibratedTangdist.txt') # p_1 and p_2
    dist.append(rDist)
    dist.append(tDist)
    dist = np.asarray(dist).reshape(1,4)

    #Load rotational and translational errors
    rotVecError = np.loadtxt('rotVecError.txt')
    transVecError = np.loadtxt('transVecError.txt')

    return ret,rets,mtx,tvecs,rMats,dist,rotVecError,transVecError
```

A 1: Import Matlab-generated text-files to Python

```python
import numpy as np
from loadCali import loadCaliParam
import cv2 as cv2
import time
def triang(pix):
    t = time.time()
    ret,rets,K,tvecs,rMats,dist,rotVecError,transVecError = loadCaliParam()
    laser = np.load('C:/Users/espen/OneDrive/Master/Laserplankalibrering/
       Bilder_og_koordinater/laserplaneQP_W.npy')

    #1 Undistort the pixels
    pix = pix.reshape(-2, 1, 2)
    undistorted_pixels = cv2.undistortPoints(pix, K, dist).reshape(-1,2)

    #2 Triangulate
    ext_points = np.array([])

    #Define lasernormal and laserpoint
    ln = laser[0:3]
    lp = laser[3:6]

    l_0 = np.array([0,0,0])

    for coord in undistorted_pixels:
        coord = np.append(coord,1)
        l = coord/np.linalg.norm(coord)
        num = np.dot((lp - l_0), ln)
        deno = np.dot(l,ln)
        d = num/deno
        fullcoord = np.array([l * d]) + l_0
        ext_points = np.append(ext_points,fullcoord)
    elapsed = time.time() - t

    return np.reshape(ext_points,(-1,3))
```

A 2: Code for triangulation

```
from AT_cx_functions import*
import cx.cx_cam as cam
import cx.cx_base as base
import time
from matplotlib import pyplot as plt
import cv2
import os
import glob

#Find and configure the device
hDev = getDevice()
cogConfig(hDev)

wd = getCurrentWD()
laserimage_folder = wd + "/laserimage"
if not os.path.exists(laserimage_folder):
    os.makedirs(laserimage_folder)
    print("Path made")
os.chdir(laserimage_folder)
#Snap an image of the scene in COG mode and save the output.
laserlinepixels = snap(hDev)
laserpixel = pixCoordify(laserlinepixels.ravel() / 64 , 2048) #reshape the
   COG-image from (2048,) to (2048,2)
laserPixelList = os.getcwd().replace("\\","/") + "/*.npy"
laserImageList = os.getcwd().replace("\\","/") + "/*.png"
laserPixelList = glob.glob(laserPixelList)
laserImageList = glob.glob(laserImageList)
num_img = len(laserImageList)
img_thresh = 10 #number of images needed for a good calibration, n < 10
if laserPixelList == []:
    save_name = wd + "/laserimage/pixcoord_1.npy"
    np.save(save_name,laserpixel)
else:
    n = len(laserPixelList) + 1
    save_name = wd + "/laserimage/pixcoord_" + str(n) + ".npy"
    np.save(save_name,laserpixel)
#Snap an image in image mode and save it
if cam.cx_getParam(hDev,"CameraMode")[1] != "Image":
    imageConfig(hDev)
    time.sleep(5)
    if laserImageList == []:
        save_name = laserimage_folder + "/1.png"
        image = snap(hDev)
        print("Image taken")
        cv2.imwrite(save_name,image)
    elif num_img < img_thresh:
        print("\nNeed "+ str((img_thresh-num_img)) + " more pictures of
           chessboard")
        #copy the images already in the folder to a list
        images = []
        for fname in laserImageList:
            images.append(cv2.imread(fname))
        image = snap(hDev)
        images.append(image)
        n = 1
        for img in images:
            save_name = laserimage_folder + "/" + str(n) + ".png"
            cv2.imwrite(save_name, img)
            n = n + 1
    if cam.cx_getParam(hDev,"CameraMode")[1] != "CenterOfGravity":
        cogConfig(hDev)
```

A 3: Code for image and CoG-scan

```
#include <Adafruit_ADS1015.h>
#include <Wire.h>
Adafruit_ADS1015 ads1015;
int analogInput = 0;
int val;
int16_t res;
float angle_analog;
float angle_digital;
void setup(){
   pinMode(analogInput, INPUT); //assigning the input port
   Serial.begin(9600); //BaudRate
   ads1015.setGain(GAIN_TWO); //Set appropriate gain
   ads1015.begin(); //Initiate the ADC
}
void loop(){
   res = ads1015.readADC_Differential_0_1(); //Differential mode between A0
       and A1
   val = analogRead(analogInput);//reads the analog input
   angle_analog = map(val, 104,921 , 0, 35999);
   angle_digital = map(res, 1970,-1963 , 0, 35999);

   Serial.print("Analog angle ");
   Serial.print(angle_analog/100);
   Serial.println(" ");
   Serial.print("Digital angle ");
   Serial.print(angle_digital/100);

   delay(20);
}
```

A 4: Arduino code for comparison

```
1
2
3  // This is very similar to Example 3 - Receive with start- and end-markers
4  //    in Serial Input Basics   http://forum.arduino.cc/index.php?topic
       =396450.0
5  #include <Adafruit_ADS1015.h>
6  #include <Wire.h>
7  int analogInput = 0;
8  float angle;
9  const byte numChars = 64;
10 char receivedChars[numChars];
11 int16_t results;
12 boolean newData = false;
13 Adafruit_ADS1015 ads1015;
14 byte ledPin = 13;   // the onboard LED
15
16 //===============
17
18 void setup() {
19     Serial.begin(115200);
20     ads1015.setGain(GAIN_TWO);
21     pinMode(ledPin, OUTPUT);
22     digitalWrite(ledPin, HIGH);
23     delay(200);
24     digitalWrite(ledPin, LOW);
25     delay(200);
26     digitalWrite(ledPin, HIGH);
27     ads1015.begin();
28     Serial.println("<Arduino is ready>");
29
30 }
31
32 //===============
33
34 void loop() {
35     recvWithStartEndMarkers();
36     replyToPython();
37 }
38
39 //===============
40
41
42 //===============
```

A 5: Robin2 Arduino code: Part 1

```
void recvWithStartEndMarkers() {
    static boolean recvInProgress = false;
    static byte ndx = 0;
    char startMarker = '<';
    char endMarker = '>';
    char rc;

    while (Serial.available() > 0 && newData == false) {
        rc = Serial.read();

        if (recvInProgress == true) {
            if (rc != endMarker) {
                receivedChars[ndx] = rc;
                ndx++;
                if (ndx >= numChars) {
                    ndx = numChars - 1;
                }
            }
            else {
                receivedChars[ndx] = '\0'; // terminate the string
                recvInProgress = false;
                ndx = 0;
                newData = true;
            }
        }

        else if (rc == startMarker) {
            recvInProgress = true;
        }
    }
}

//===============

void replyToPython() {
    results = ads1015.readADC_Differential_0_1(); //Differential mode
       between port A0 and A1
    angle = map(results, 1970, -1963, 0, 35999); // Map the counts to
       values between 0 and 35999


    if (newData == true) {
        Serial.print("<Angular reading:");
        Serial.print(" ");
        Serial.print(angle/100);
        Serial.print(" ");
        Serial.print('>');
        digitalWrite(ledPin, ! digitalRead(ledPin));
        newData = false;
    }
}
```

A 6: Robin2 Arduino code: Part 2

```
import serial
import time

startMarker = '<'
endMarker = '>'
dataStarted = False
dataBuf = ""
messageComplete = False

def setupSerial(baudRate, serialPortName):
    global  serialPort
    serialPort = serial.Serial(port= serialPortName, baudrate = baudRate,
        timeout=0, rtscts=True)
    print("Serial port " + serialPortName + " opened  Baudrate " + str(
        baudRate))
    waitForArduino()
def sendToArduino(stringToSend):
        # this adds the start- and end-markers before sending
    global startMarker, endMarker, serialPort

    stringWithMarkers = (startMarker)
    stringWithMarkers += stringToSend
    stringWithMarkers += (endMarker)

    serialPort.write(stringWithMarkers.encode('utf-8')) # encode needed for
        Python3
def recvLikeArduino():
    global startMarker, endMarker, serialPort, dataStarted, dataBuf,
        messageComplete
    if serialPort.inWaiting() > 0 and messageComplete == False:
        x = serialPort.read().decode("utf-8") # decode needed for Python3
        if dataStarted == True:
            if x != endMarker:
                dataBuf = dataBuf + x
            else:
                dataStarted = False
                messageComplete = True
        elif x == startMarker:
            dataBuf = ''
            dataStarted = True

    if (messageComplete == True):
        messageComplete = False
        return dataBuf
    else:
        return "XXX"
def waitForArduino():
    print("Waiting for Arduino to reset")
    msg = ""
    while msg.find("Arduino is ready") == -1:
        msg = recvLikeArduino()
        if not (msg == 'XXX'):
            print(msg)
def CollectAngle():
    sendToArduino("Reqesting reading")
    arduinoReply = recvLikeArduino()
    while (arduinoReply == 'XXX'):
        arduinoReply = recvLikeArduino()
        if arduinoReply != 'XXX':
            return (arduinoReply.split(" ")[2])
```

A 7: Robin2 Python code

```
ret,rets,K,tvecs,rMats,dist,rotVecError,transVecError = loadCaliParam()
os.chdir('C:/Users/espen/OneDrive/Master/Laserplankalibrering/
    Bilder_og_koordinater/LaserAndNpys')
laser_npy = os.getcwd().replace("\\","/") + "/*.npy"
laser_npy = glob.glob(laser_npy)
number_of_laserfiles = len(laser_npy)
ext_points = np.array([])
j = 0
for i in range(1,len(laser_npy)+1):

    RotM = rMats[j]
    print(i)
    tVec = tvecs[j]
    n = RotM[2,:]
    p_0 = tVec
    l_0 = np.array([0,0,0])
    filename = 'pixcoord_' + str(i) + '.npy'
    pix_coord = np.load(filename)
    k = 0
    pure_coords = np.array([])
    for reading in pix_coord:
        if reading[1] != 0 and 650<=k<=1700 : #Remove pixels due to
            interference, the second condition can be removed if the images
            are clean
                pix = [reading[0],reading[1]]
                pure_coords = np.append(pure_coords,pix)

        k += 1
    pix_coord = pure_coords.reshape(-2, 1, 2)

    undistorted_point = cv2.undistortPoints(pix_coord, K, dist).reshape
        (-1,2)

    for coord in undistorted_point:
        coord = np.append(coord,1)
        l = coord/np.linalg.norm(coord)
        num = np.dot((p_0 - l_0), n)
        deno = np.dot(l,n)
        d = num/deno
        fullcoord = np.array([l * d]) + l_0
        ext_points = np.append(ext_points,fullcoord)

    j = j + 1

ext_points = np.reshape(ext_points,(-1,3))
```

A 8: Generation of point cloud

```
x = ext_points[:,0]
y = ext_points[:,1]
z = ext_points[:,2]

A = np.empty((len(x),3))
b = np.empty((len(x),1))

b[:,0] = -z
A[:,0] = x
A[:,1] = y
A[:,2] = 1


solution = np.linalg.inv((np.transpose(A) @ A)) @ np.transpose(A) @ b
print('sol',solution)

plane_vec = [float(solution[0]), float(solution[1]), 1]
norm_plane_vec = plane_vec
point_on_plane = [0,0,-float(solution[2])]

plane_representation = [norm_plane_vec[0],norm_plane_vec[1],norm_plane_vec
    [2],point_on_plane[0],point_on_plane[1],point_on_plane[2]]
```

A 9: Least squares

```python
import numpy as np
import os
import glob
from scipy.optimize import minimize
from weightAssigner import weights

def opt_solver(dictio,W):
    #Define Centroid of pointcloud
    x_values = np.array([])
    y_values = np.array([])
    z_values = np.array([])
    for key,val in dictio.items():
        x_values = np.append(x_values,val[0::3])
        y_values = np.append(y_values,val[1::3])
        z_values = np.append(z_values,val[2::3])

    [cx_, cy_, cz_] = np.array([sum(x_values)/len(x_values),sum(y_values)/
       len(y_values),sum(z_values)/len(z_values)])

    #Distance from each point to centroid
    centroid_vectors = {}
    for key,val in dictio.items():
        centroid_vectors[key] = np.array([(val[0::3]-cx_),(val[1::3]-cy_),(
           val[2::3]-cz_)])

    def cost_func_W(n):
        summ = 0
        for key,val in centroid_vectors.items():
            for k in range(len(val[1,:])):

                summ += W[key]*(val[0,k]*n[0] + val[1,k]*n[1] + val[2,k]*n
                   [2])**2
        return summ
    def cost_func(n):
        summ = 0
        for key,val in centroid_vectors.items():
            for k in range(len(val[1,:])):

                summ += (val[0,k]*n[0] + val[1,k]*n[1] + val[2,k]*n[2])**2
        return summ
    def constraint(n):
        return 1 - n[0]**2 - n[1]**2 - n[2]**2 #Vectorlength = 1
    const = ({'type': 'eq', 'fun':constraint})

    n0 = np.array([1,1,1])/np.sqrt(3) #Initial guess of n-vector
    if W:
        sol = minimize(cost_func_W,n0,method='SLSQP',constraints=const,
           options={'disp':True})
        print("Solving with weights")
    else:
        sol = minimize(cost_func,n0,method='SLSQP',constraints=const,
           options={'disp':True})
        print("Solving without weights")

    return sol,cx_,cy_,cz_
```

A 10: Optimization solver

```
#Import values from calibration process
ret,rets,K,tvecs,rMats,dist,rotVecError,transVecError = loadCaliParam()

K_inv = np.linalg.inv(K)
os.chdir('C:/Users/espen/OneDrive/Master/Laserplankalibrering/
    Bilder_og_koordinater/LaserAndNpys')
laser_npy = os.getcwd().replace("\\","/") + "/*.npy"
laser_npy = glob.glob(laser_npy)
number_of_laserfiles = len(laser_npy)
#print(number_of_laserfiles)
ext_points = np.array([])
j = 0
pixel_dict = {} #Sorts each extracted point in a dictionary based on image
    such that weights can be correctly assigned
for i in range(1,len(laser_npy)+1):
    pixel_dict.update({str(i):np.array([])})
    RotM = rMats[j]
    tVec = tvecs[j]
    n = RotM[2,:]
    p_0 = tVec
    l_0 = np.array([0,0,0])
    filename = 'pixcoord_' + str(i) + '.npy'
    pix_coord = np.load(filename)
    k = 0
    pure_coords = np.array([])
    for reading in pix_coord:
        if reading[1] != 0 and 650<=k<=1700 : #Remove pixels due to faulty
            points
                pix = [reading[0],reading[1]]
                pure_coords = np.append(pure_coords,pix)
        k += 1

    pix_coord = pure_coords.reshape(-2, 1, 2)
    #Undistort the pixels from the file
    undistorted_point = cv2.undistortPoints(pix_coord, K, dist).reshape
        (-1,2)
    for coord in undistorted_point:
        coord = np.append(coord,1)
        l = coord/np.linalg.norm(coord)
        num = np.dot((p_0 - l_0), n)
        deno = np.dot(l,n)
        d = num/deno
        fullcoord = np.array([l * d]) + l_0
        ext_points = np.append(ext_points,fullcoord)
        pixel_dict[str(i)] = np.append(pixel_dict[str(i)],fullcoord)
    j = j + 1
```

A 11: Pointcloud with dictionary

```
1
2  import numpy as np
3
4  def weights(Errors1,Errors2):
5      NumberOfImages = int(len(Errors1[:,0]))
6      temp_weights = {}
7      weights = {}
8      for i in range(len(Errors2[:,0])):
9          error = np.sqrt(Errors1[i,0]**2 + Errors1[i,1]**2 + Errors1[i
               ,2]**2)
10         temp_weights[str(i+1)] = error
11
12
13     for i in range(len(Errors2[:,0])):
14         t_error = np.sqrt(Errors2[i,0]**2 + Errors2[i,1]**2 + Errors2[i
               ,2]**2)
15         temp_weights[str(i+1)] += t_error
16
17     for key,val in temp_weights.items():
18         weights[key] = 1 / (val)
19     return weights
```

A 12: Function for calculating weights

```python
import numpy as np
from loadCali import loadCaliParam
import cv2 as cv2

def triang(pix):
    ret,rets,K,tvecs,rMats,dist,rotVecError,transVecError = loadCaliParam()


    #Switch between the different estimated laserplanes , point-normal notation
    laser = np.load('C:/Users/espen/OneDrive/Master/Laserplankalibrering/Bilder_og_koordinater/laserplane_QP.npy')
    #laser = np.load('C:/Users/espen/OneDrive/Master/Laserplankalibrering/Bilder_og_koordinater/laserplane_QPNW.npy')
    #laser = np.load('C:/Users/espen/OneDrive/Master/Laserplankalibrering/Bilder_og_koordinater/laserplane_LS.npy')


    #1 Remove pixels without any registered laser
    coords = np.array([])

    for pixel_coord in pix:
        if pixel_coord[1] != 0:
            coords = np.append(coords,[pixel_coord])

    coords = coords.reshape(-1,2)
    pix = coords

    #2 Undistort the pixels
    pix = pix.reshape(-2, 1, 2)
    undistorted_pixels = cv2.undistortPoints(pix, K, dist).reshape(-1,2)
    #3 Triangulate
    ext_points = np.array([])

    #Define lasernormal and laserpoint
    ln = laser[0:3] #Laser normal
    lp = laser[3:6] #Laser point

    l_0 = np.array([0,0,0])

    for coord in undistorted_pixels:
        coord = np.append(coord,1)
        l = coord/np.linalg.norm(coord)
        num = np.dot((lp - l_0), ln)
        deno = np.dot(l,ln)
        d = num/deno
        fullcoord = np.array([l * d]) + l_0
        ext_points = np.append(ext_points,fullcoord)

    return np.reshape(ext_points,(-1,3))
```

A 13: Triangulation-function for scanning

```
def planeify(vector_plane):
    if len(vector_plane) == 6: #For input [nx,ny,nz,cx,cy,cz]
        #[A,B,C,D] from Ax + By + Cz + D = 0
        plane = [vector_plane[0],vector_plane[1],vector_plane[2],-
            vector_plane[0]*(vector_plane[3])-vector_plane[1]*(vector_plane
            [4])-vector_plane[2]*(vector_plane[5])]
        #Or on form [Ax + By + D] = z
        plane_s = [-plane[0]/plane[2],-plane[1]/plane[2],-plane[3]/plane
            [2]]
        return plane,plane_s
    if len(vector_plane) == 4: #For input [A,B,C,D]
        n = [vector_plane[0],vector_plane[1],vector_plane[2]]
        d = [0,0,-vector_plane[3]/vector_plane[2]]
        return n,d
    else:
        print("Unknown representation. Check input.")
```

A 14: Converting from point-normal to standard plane representation

```
from AT_cx_functions import*
from TriangulateReading import triang
import cv2 as cv2
import serial
import time
import numpy as np
import os
from arduinoComm import *
import serial
import msvcrt

#Find and configure the camera
hDev = getDevice()
cogConfig(hDev)

#Establish connection with Arduino
setupSerial(115200, "COM3")
#COMX depends on which USB-port the Arduino is connected to

#Create dictionary with angles as key and pixels as values
data = {}
#Variable for terminating the scanning process
k = 'dummy'

while True:
    t1 = time.time()
    #Snap an image of the scene in COG mode.
    laserlinepixels = snap(hDev)
    #reshape the COG-image from (2048,) to (2048,2)
    laserpixel = pixCoordify(laserlinepixels.ravel() / 64 , 2048)

    dict_key = CollectAngle()
    data[dict_key] = laserpixel
    t = time.time() - t1

    #These conditions enabled manual termination of the scan
    if msvcrt.kbhit():
        k = str(msvcrt.getch()).replace("b'","").replace("'","")
    if k == 'q':
        print("Scanning ended")
        break

os.chdir('C:/Users/espen/OneDrive/Master/Laserplankalibrering/
   Bilder_og_koordinater/Extracted Readings')
np.save('angleandlaserreading.npy',data)
print('Readings saved')
```

A 15: Program for scanning object

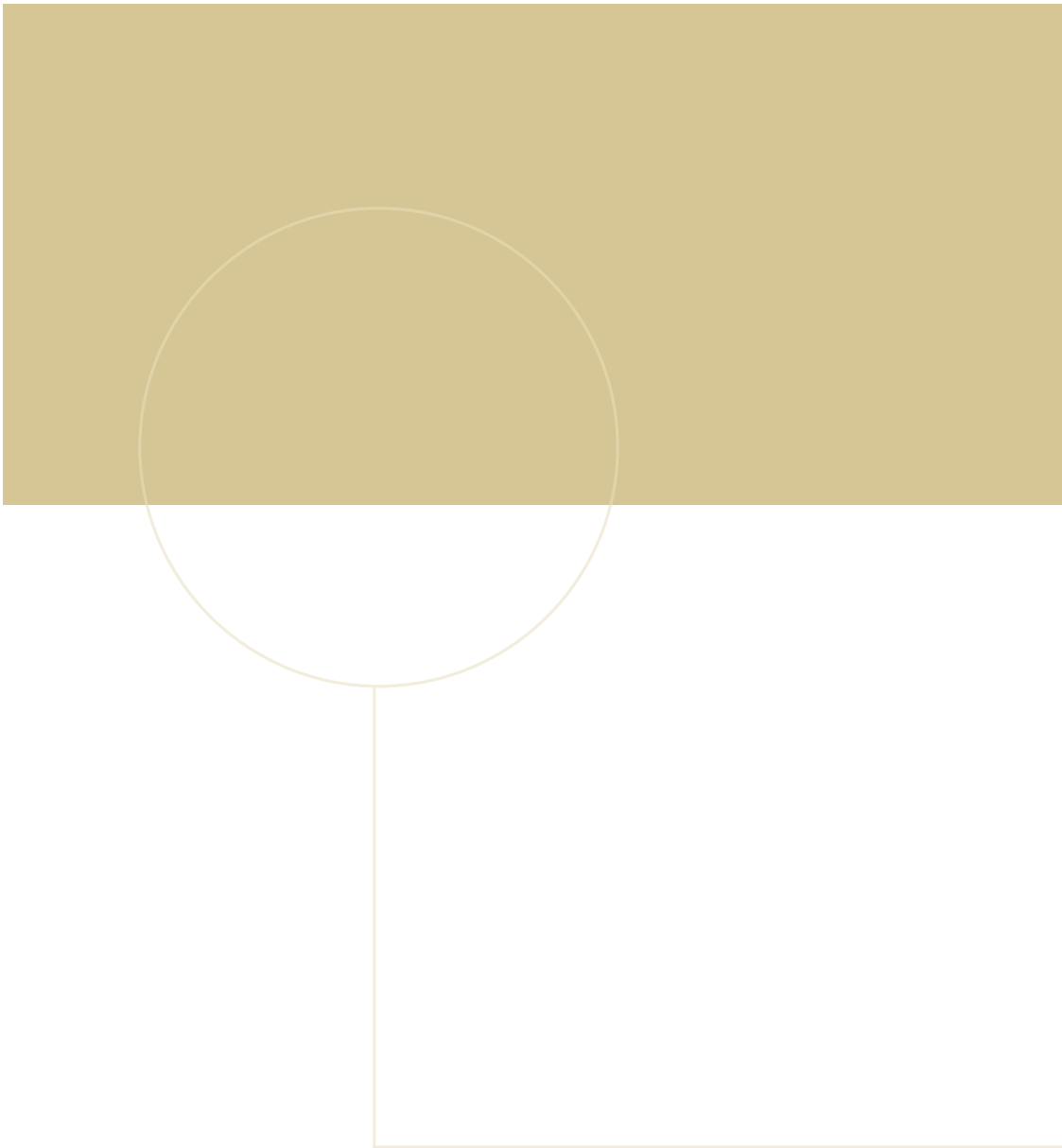
```
import numpy as np
import os
from scipy import polyfit
import numpy as np
from matplotlib import pyplot as plt

os.chdir('C:/Users/espen/OneDrive/Master/Laserplankalibrering/
    Bilder_og_koordinater')
data = np.load('pcOffsetPlaneTrans.npy')

epsi = 0.075
groove = -7.0901
dist_min = groove - epsi
dist_max =  groove + epsi

path1 = np.empty((50000000,3))

k = 0

for points in data:
    if points[0] >=  dist_min and points[0] <= dist_max:
        path1[k,:] = points
        k += 1
path = path1[0:k,:]
avgX = sum(path[:,0])/len(path[:,0])
print(avgX)

np.delete(path,path[k:,:])
x = path[:,1]
y = path[:,2]
plt.scatter(x,y, label='Datapoints')
plt.ylabel("z-coordinate [mm]")
plt.xlabel("Angle [\xb0]")
plt.xticks(np.arange(0, 370, 30))
plt.yticks(np.arange(int(min(y)), int(max(y))+5, 2))



#Poly fit
number_of_points = len(x)
linsp = np.linspace(0, 360, 360)

[fit,resi,_,_,_] = np.polyfit(x, y, 8, rcond=None, full=True, w=None, cov=
    False)
print(resi/number_of_points)
p = np.poly1d(fit)
print(p)

plt.plot(p(linsp),'-',color='r', label='Polyfit deg=8')
plt.legend(loc='upper right')
plt.show()
```

A 16: Polyfit

```
import numpy as np
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import os
from TriangulateReading import triang
import time
from decimal import Decimal


dir_path = os.path.dirname(os.path.realpath(__file__)).replace("\\", "/")
os.chdir(dir_path + '/TestResults/Rotaryscan')
#data = np.load('angleandlaserreading.npy').item() #Centric Test
data = np.load('angleandlaserOffset.npy').item() #Non-centric Test
keys = []
#Convert angle-strings to actual numbers
for key in data:
    keys.append("{:.2f}".format(float(key)))

low_high_keys = sorted(keys)

#Construct uniform format to angular reading
for name in low_high_keys:
    isShort = str(name).split('.')
    if len(isShort[1]) == 1:
        real_name = str(name) + str(0)
T_camChess = np.array([[0.9993,-0.0091,0.0359,-63.126109659696326],
                   [0.0203,0.9454,-0.3252,-69.662268476300030],
                   [-0.0310,0.3257,0.9450,541.8121286398493],
                   [0,0,0,1]])

cos = np.cos
sin = np.sin
#This function converts the points to the frame of the encoder
def convertPipe():
    transPoints = np.empty([1525525,3])
    cameraPoints = np.empty([1525525,3])
    max_dist = 0
    min_dist = 1000
    count = 0

    for key in low_high_keys:
        rad = float(key) * np.pi/180
        pixels = data[str(key)]
        coord_points = triang(pixels)



        T_chessEnc = np.array([[1,0,0,0],
                    [0,cos(rad),-sin(rad),30],
                    [0,sin(rad),cos(rad),0],
                    [0,0,0,1]])
        T_camEnc = T_camChess @ T_chessEnc
        T_encCam = np.linalg.inv(T_camEnc)

        for point in coord_points:
            transPoint = (T_encCam @ [point[0],point[1],point[2],1])[0:3]
            if point[2] >= max_dist:
                max_dist = point[2]
            if point[2] <= min_dist:
                min_dist = point[2]
            transPoints[count,:] = [transPoint[0],transPoint[1],transPoint
                [2]]
```

```
#This function transforms the readings to a frame located approximately at
    the plane of the laser
def convertPlane():
    transPoints = np.empty([1525525,3])
    count = 0
    TransZ = np.array([[1,0,0,0],
            [0,1,0,0],
            [0,0,1,589.48137],
            [0,0,0,1]])
    RotX = np.array([[1,0,0,0],
        [0,cos(-21.12 * np.pi/180),-sin(-21.12 * np.pi/180),0],
        [0,sin(-21.12 * np.pi/180),cos(-21.12 * np.pi/180),0],
        [0,0,0,1]])
    TransZ2 = np.array([[1,0,0,0],
            [0,1,0,0],
            [0,0,1,-549.8846186],
            [0,0,0,1]])
    Trans = TransZ @ RotX @ TransZ2
    T_plan = np.linalg.inv(Trans)
    for key in low_high_keys:
        rad = float(key)
        pixels = data[str(key)]
        coord_points = triang(pixels)
        TransY = np.array([[1,0,0,0],
            [0,1,0,rad],
            [0,0,1,0],
            [0,0,0,1]])
        PointTrans = TransY @ T_plan

        for point in coord_points:
            transPoint = (PointTrans @ [point[0],point[1],point[2],1])[0:3]
            transPoints[count,:] = [transPoint[0],transPoint[1],transPoint
                [2]]
            count += 1
    return transPoints
clean_points = convertPlane()
#np.save('pcOffsetPlaneTrans.npy',clean_points)
print("File saved")
```

A 18: Homogeneous transformation of pointcloud : Laserplane



NTNU
Norwegian University of
Science and Technology