

TTK4551 Specialization Project

Using computer vision to control a robotic welder



Kunnskap for en bedre verden

Department of Engineering Cybernetics

Supervisor: Ole Morten Aamo, ITK

Kristoffer Hermansen

June 10, 2019

Contents

1	Introduction	3
2	Theoretical Background	4
2.1	OpenCV	4
2.2	Noise reduction	4
2.3	Edge detection	4
2.4	Transformations	5
2.5	Optics	6
3	Method	7
3.1	Computer Vision	7
3.2	Feature detection	8
3.3	3D-environment	8
3.4	Camera calibration	10
3.5	Transformations	12
4	Results	14
4.1	Image processing	14
4.2	Detecting features	15
4.3	Estimated weld start	19
5	Discussion	20
5.1	Image processing	20
5.2	Detecting features	20
5.3	Setup	20
5.4	Transformation	21
6	Conclusion and future work	22
6.1	Conclusion	22
6.2	Future work	22
	Appendices	24
A	Python code	25
A.1	Main code	25
A.2	Transformation	28
A.3	Camera calibration	30

List of Figures

3.1	Window with trackbar for easy parameter tuning of the Gaussian blur	7
3.2	Smooth, textured and real drill pipe respectively	8
3.3	Overall view of 3D-modeled system	9
3.4	Inside view of 3D-modeled system	9
3.5	Camera frame to start-point	10
3.6	3D-modeled chessboard	11
3.7	6:9 chess pattern recognized	11
3.8	Some of the images used for the camera calibration, showing the input image and image after recognizing the chessboards	12
4.1	Drill pipe with smooth surface	14
4.2	Drill pipe with textured surface	14
4.3	Real drill pipe	15
4.4	Drill pipe with smooth surface, threshold = 0.38	15
4.5	Drill pipe with textured surface, threshold = 0.38	15
4.6	Real drill pipe, threshold = 0.45. Note that shadow from the weld gun made the feature detection on the right side of pipe to difficult for the current detector. Therefore another feature detection image was used, which was the other quarter of a circle, which is why the detected weld start is on the other side of the groove.	16
4.7	Result overlaid to original image	16
4.8	Original image from inventor-camera, inventor zoom angle at 60 degrees	17
4.9	Detected start-point for welding, overlayed on original image	17
4.10	zoomed result	18

Chapter 1

Introduction

This paper is being written on behalf of WellConnection Mongstad. WellConnection Mongstad is a supplier of inspection, maintenance and repair (*IMR*) services on equipment for the oil and gas industry. A substantial part of WellConnections work is related to *IMR* of drill pipes. The *IMR* work on the drill pipes consists of several parts, and vary a lot depending on the state of the pipes received. The work initially consists of cleaning the pipes before inspection, then the amount of work required on maintenance and repair is decided on behalf of the inspection results. After this the pipes are sent through the automatic production line where all of the maintenance and repair work is handled.

For this report, the main interest is the repair of the soft-legging layer under the hard-banding welds. When the pipes are being mounted together, while drilling and when taken apart, a lot of wear and tear happens on the tool joints. The tool joint is the section on the drill pipe with the enlarged diameter and where iron roughneck attaches when screwing them together during operations. In order to extend the lifetime of the drill pipe, repairing the hard-banding weld is essential. The pipes are first sent to a lathe for machining to remove the old hard-banding. Sometimes it is enough to just remove the old layer and then weld a new hardbanding layer. Other times the pipe has deeper pores under the weld, which require a larger diameter to be removed until the surface is smooth and free of pores.

When the deeper pores are being removed, the diameter of the tool joint will decrease. Therefore the drill pipe needs to have another layer of weld to get the diameter back up before the hard-banding is welded. This layer of weld is called soft-lagging, which is a softer weld more similar to the quality of the pipe. Welding the soft-lagging layer is time consuming due to the large amount of welding needed, and a temperature restriction in order to not burn the coating inside the drill pipes.

With this in mind, WellConnection is therefore interested in creating an additional production line which can handle the soft-lagging pipes such that they do not cause delays for the main production line. In any large project, there are numerous ways to the end goal. In pursuit for the most beneficial solution, this paper will, together with another paper "*Utilizing laser triangulation to extract spatial coordinates of drill-pipe's geometry*"(Liavik (2018)) try to answer the most critical questions at an early stage. To look at the possibilities for creating as much of the production line automated, a few key questions needs to be answered. In the existing welding stations, manual chucks are used to clamp the pipes to allow for rotation of the pipes. If this process is to be automated, there are substantial costs to the industry standard pneumatic clamping chucks, therefore we wanted to make sure that it would be possible to automate the control of the welding robot before the decision on clamping was chosen. If the welding process can not be automated, there will be few benefits with the more expensive pneumatic chuck. Since it needs to be an operator to control the welding robot either way, the operator could then easily operate the manual chuck in addition to the welding.

Problem definition:

Create a program which can be used to identify where the soft-lagging weld should be started using computer vision. Furthermore use the pixel coordinates to project the starting point of the weld to the 3D coordinates for the robot arm. This information is then to be used as control input for the robot arm.

This is not intended to be a scientific paper on how to make the best edge detector and find to most efficiently methods of interpreting this to generate 3D-coordinates. This report is however intended to solve a real problem and make it possible for WellConnection to have the most benefit of my time. The focus will be on proving concepts and trying to implement these concepts to some degree, making it as realizable as possible for the company.

Chapter 2

Theoretical Background

2.1 OpenCV

In this paper OpenCV is used as the main library for computer vision. OpenCV is, in a more general term; "an open source computer vision and machine learning software library". OpenCV has a python interface and supports all the major operating systems. In this report the library is used to save a lot of time by using well established functions. Since OpenCV is open source, all the code can be further improved and tuned later without the need for any licenses. (OpenCV team (2018))

2.2 Noise reduction

Prior to the actual blurring, the image is read as a gray scale image. In order to detect edges in an image, noise reduction (smoothing or blurring) the image is essential to get good results. Without the initial blurring the edge detector will struggle to see the difference between textures and edges, making the result less than helpful. Blurring an image can be viewed as using a low pass filter on the image, since it reduces the noise in the image. There are several ways to blur an image, after looking at the mean filter method, median filter method and the Gaussian filter method, the latter gave the better results.

The Gaussian filter looks at each pixel in the original image, and then the kernel used takes the surrounding pixels and tries to make a weighted average which becomes the new pixel value for the blurred image. It is the ability to easily prioritize the closest pixels more than the pixels further away that makes the Gaussian blur so applicable for use before edge detection. This Gaussian weight in the kernel does a better job preserving potential edges, which is why this method ended up being the preferred method for image blurring in this report. (OpenCV team (2015))

2.3 Edge detection

To detect edges, the general concept is to apply a high pass filter to detect changes in neighbouring pixels, which will be interpreted as edges. This can be done with a custom kernel which enlarges pixel values with large gradients, indicating the presence of an edge. However, tuning this kernel will not create a robust edge detector, and several steps are required to create an usable edge detector algorithm. There are several algorithms and techniques to achieving edge detection in an image, the one used in this paper is the Canny Edge Detection. The canny edge detection algorithm contains four steps:

1. Noise reduction to blur the image
2. Find the intensity gradients to locate edges
3. Non-maximum suppression in order to thin the edges.
4. Hysteresis thresholding in order to delete the unqualified edges.

(Alexander Mordvintsev, Abid K (2013))

2.4 Transformations

In order to express distances from one frame, to another frame with a different orientation, the different frames need to be linked by the use of rotation matrix. From equation 2.1 we see the rotation matrices which each describe a rotation around one axis, where R_x , R_y and R_z is the rotation around the x-axis, y-axis and z-axis respectively.

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \quad R_y = \begin{bmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 0 & 1 \\ -\sin(\phi) & 0 & \cos(\phi) \end{bmatrix} \quad R_z = \begin{bmatrix} \cos(\phi) & -\sin(\phi) & 0 \\ \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.1)$$

In addition, the different frames, might be located in different places. When this is the case, one also has to take the distance between them into account when describing points from one frame to another. The translation from a point A, to a point B seen from frame A, can then be described using a translation vector $t_{AB}^A = [x, y, z]^T$. By the combination of translation and rotation, one can easily go from one frame to another and still be able to describe points and vectors from the other frame. The combination of translation and rotation can be described in one single matrix, called the Transformation matrix. (Egeland (2018a))

$$T_{AB}^A = \begin{bmatrix} R_B^A & t_{AB}^A \\ \mathbf{0} & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & \cos(\theta) & -\sin(\theta) & y \\ 0 & \sin(\theta) & \cos(\theta) & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.2)$$

In 2.2 the rotation happens to be a rotation around the x-axis in this case. The notation used for the transformation matrices is explained in equation 2.3:

$$T_{AB}^A = T_{\text{from point A to point B}}^{\text{Seen from frame A}} \quad (2.3)$$

2.5 Optics

The camera parameter matrix is a matrix containing important parameters which describe the intrinsic parameters of the camera. To obtain the intrinsic parameters, a camera calibration can be done. The result after the calibration is the camera parameter matrix, denoted K .

$$K = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \quad f_x = \frac{f}{\rho_w} \quad f_y = \frac{f}{\rho_h} \quad (2.4)$$

Where f is the focal length, ρ_w is the width of a pixel and ρ_h is the height of a pixel. Further u_0 and v_0 are the pixel coordinates at the center of the image plane. The convention also gives the origin of the image plane is the top left corner, and moving horizontally to the right is the positive u-direction, while moving vertically down is the positive v-direction. (Egeland (2018b))

$$p = \begin{bmatrix} u \\ v \end{bmatrix}, \quad \tilde{p} = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \quad (2.5)$$

Chapter 3

Method

3.1 Computer Vision

The first step prior to any image processing, was to read up on the image processing toolbox which was to be used, OpenCv. The initial work was based on getting familiarized with the functions needed to read, save and show images. After gaining a general understanding in OpenCv, the time was then spent on research on how to detect edges in images, how have other people done it earlier and then trying to make a program suitable for detecting edges on a drill pipe.

The approach used was to use a gray scale image, and an initial Gaussian filter on that image. The Canny algorithm does involve a blurring step, but the results did not create a satisfying result without an additional blurring filter, and the already blurred image was used as input for the canny edge detector. In some cases this step might not be necessary, but as rust and other surface texture caused false edges without the additional blurring, the initial blurring before the canny algorithm was used. The second step was to use a canny edge detector function from the OpenCv library. To create appropriate edge detection on the drill pipes, some tuning of both the blurring kernel and the canny parameters was necessary.

To simplify the tuning of the filters, a live tracker was added to the image window, linked to key parameters for the `cv2.GaussianBlur()` function, as well as for the parameters for `cv2.Canny()`. The use of a tracker was helpful in better understanding how the filters worked, as it enabled the ability to continuously edit variables and immediately see the difference in the image. After some tuning a suitable result was achieved.

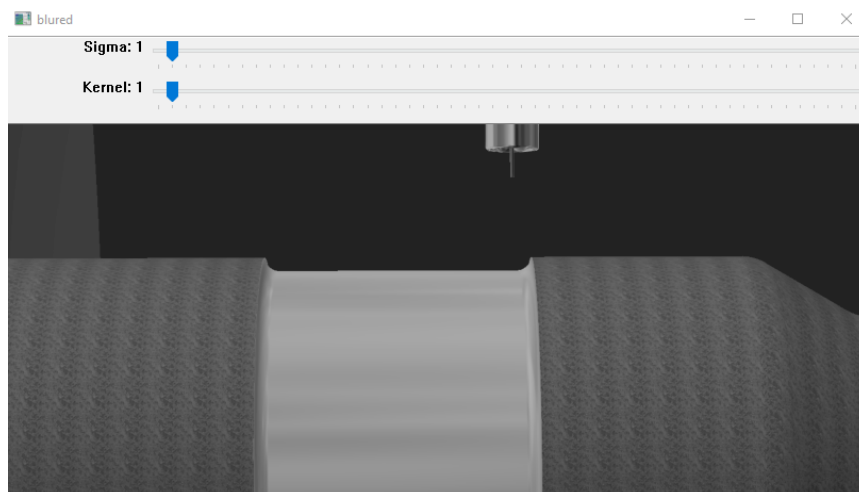


Figure 3.1: Window with tracker for easy parameter tuning of the Gaussian blur

To get suitable testing images, 3D-CAD software (Autodesk Inventor Professional 2019), hereafter called Autodesk Inventor, was used to create an environment with a drill pipe. On the drill pipe, a machined groove similar to the one made for the pipes in need of soft-lagging was created. After creating the 3D-model it was given a smooth texture in order to make the initial detecting simpler. The first program used a print screen of the pipe in the 3D-model environment to be used in the image input. After this, the 3D-model was applied a more realistic texture, to see how the program would handle detecting edges on a pipe while a texture provides unwanted edges. Finally a real image of a drill pipe in a welding station where tested. The three pipes before image processing can be seen in figure 3.2.

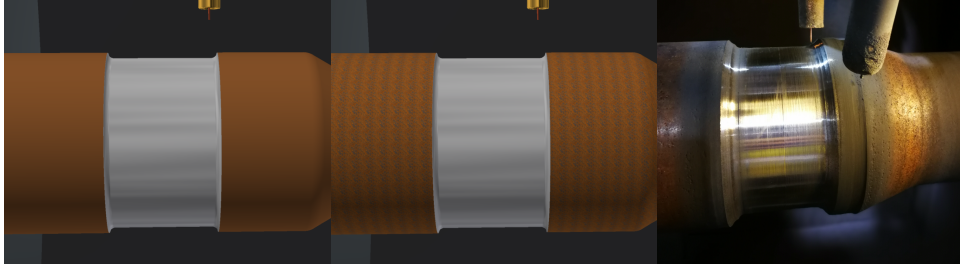


Figure 3.2: Smooth, textured and real drill pipe respectively

3.2 Feature detection

To create a simple feature detector, the first task is to figure out how the desired feature can be detected. In this case, the cutting inserts used to create the machined groove is circular to improve its strength. This led to the idea to search for a quarter of a circle, since this would represent the visible start of the machined groove. To create the detection image, an image containing a half circle was obtained, and it was processed the same way as the drill pipes to detect a white edge on a black background, similar to the processed drill pipe images. Then the `cv2.matchTemplate` function was used to look for similarities in the drill pipe image compared to the detection image. Some tuning was necessary to decide which features were actually similar. This tuning was done by using a parameter called threshold, with a value between 0 and 1.

The program then returns an image with colored rectangles at the location of the detected features. The position of the location of the feature is also returned in pixel coordinates. To improve the detection function, some logical barriers were added to prevent the program detecting features at unexpected places in the image, such as the bottom half of the image, which makes the tuning a bit easier and the program more robust.

3.3 3D-environment

After the computer vision program can detect the weld-start, the next part was to calculate the 3D position of the weld-start with the pixel coordinates generated from the detected weld-start. To do this without having any data from the location of the camera from the actual images of the drill pipes, trying to calculate the 3D location from these images did not make sense. The next step was therefore to improve the 3D model and use this model in the calculations.

The ability to use 3D models for initial testing have been essential for being able to validate the code while working. This specially yields for ensuring that the transformation from the detected start point in pixel coordinates can be transformed to 3D coordinates, since the location of both camera frame, world frame and the wanted start-point can be extracted with exact precision. The setup can be viewed from figure 3.3 and 3.4

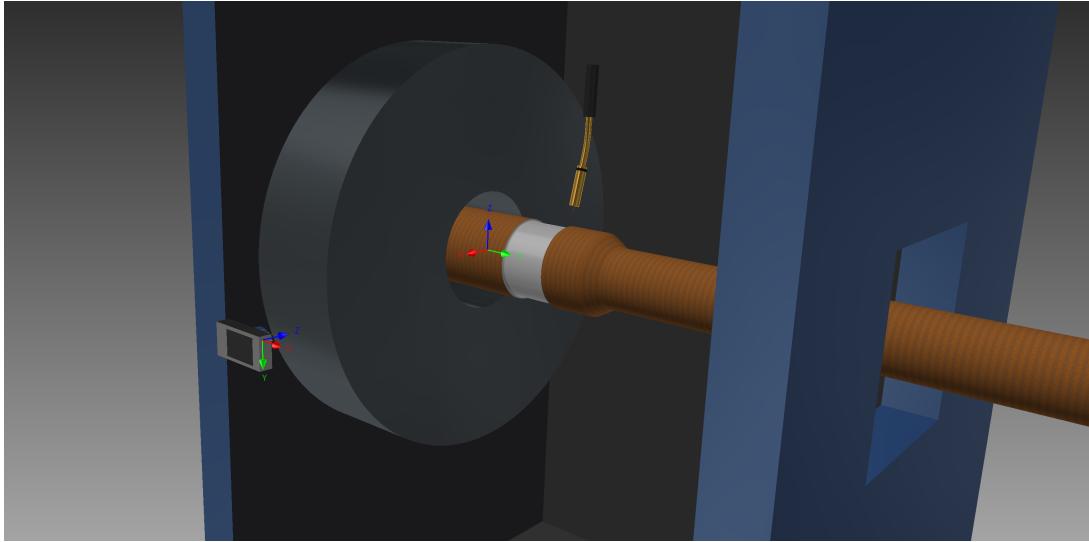


Figure 3.3: Overall view of 3D-modeled system

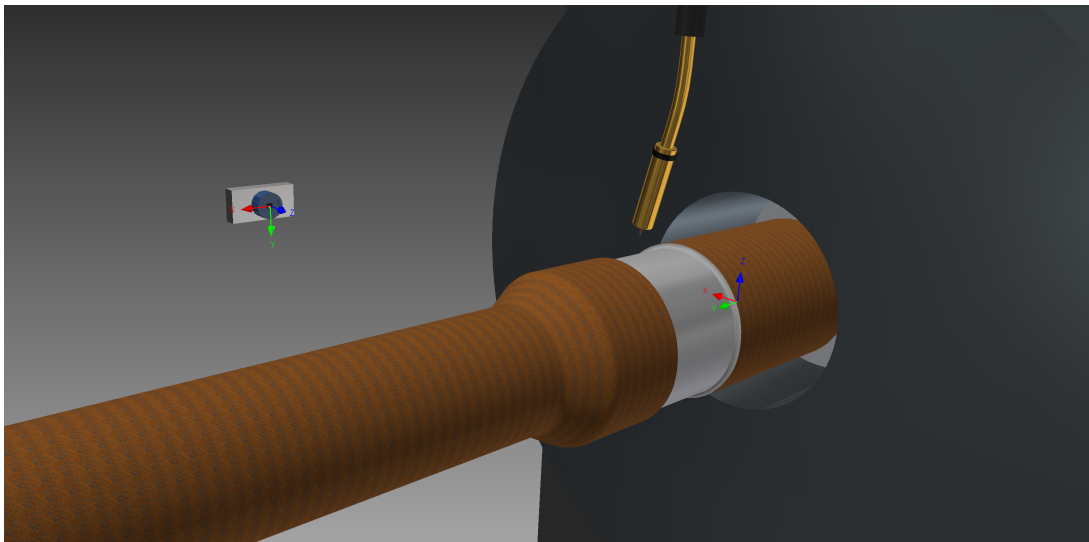


Figure 3.4: Inside view of 3D-modeled system

The distances between the frames and to the start-point have been measured in inventor and are shown in figure 3.5, and where done similarly for the rest. Note that the measurement box shows lengths from inventors default coordinate system, which has a different orientation than the one chosen for the world-frame in the machine. All the distances are also given here in their correct frame representations in the results.

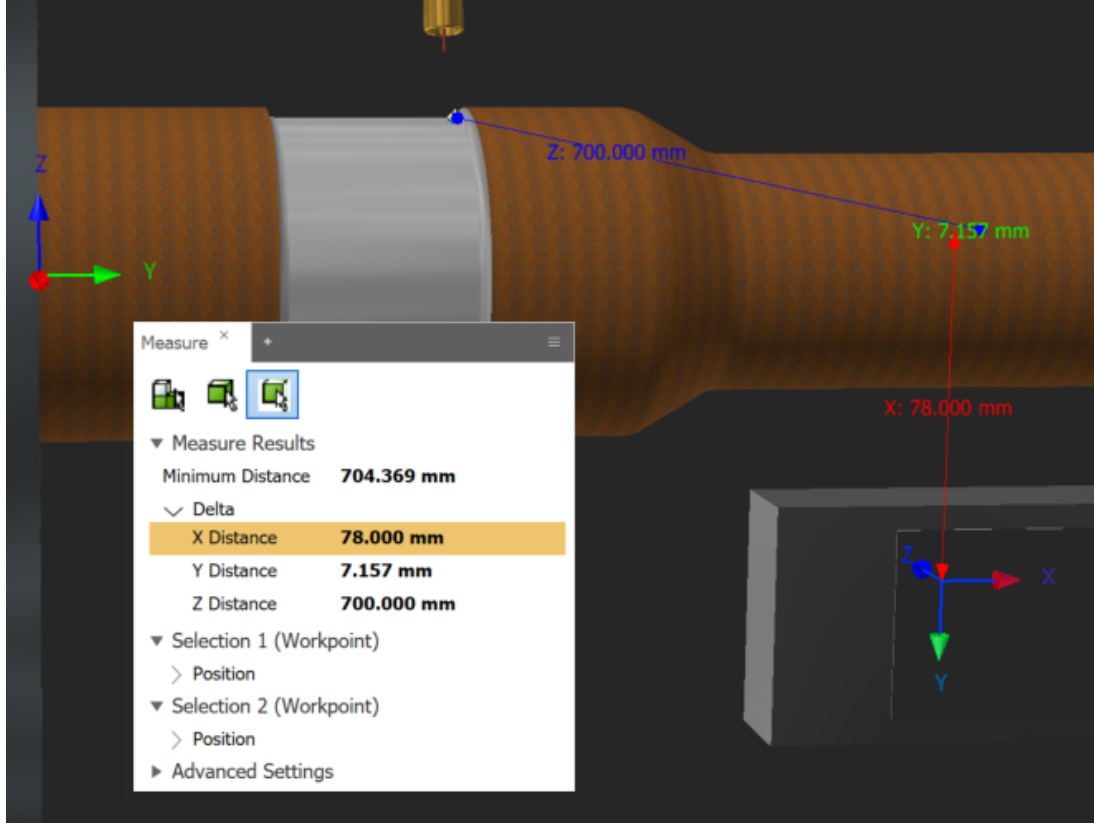


Figure 3.5: Camera frame to start-point

3.4 Camera calibration

In order to calculate any positions using the images taken for this project, we need to know the camera calibration matrix from the camera used for the images. In this case the images used was saved images from Inventor, and therefore the images used for the calibration also had to be saved in the same way from inventor. The calibration chessboard was created in Autodesk inventor, a print screen of the 3D-model can be seen from figure 3.6. To find the camera calibration matrix, a total of 20 images saved from different angles was used. Further these images was run through a calibration code used to locate the chessboard 6:9 size in this case. Figure 3.7 shows the pattern recognized by the calibration program. Interpreting these findings and using `cv2.calibrateCamera` OpenCv function, a calibration matrix K , was derived. The K -matrix had the usual form as seen in equation 3.1, and the actual numbers are given in the results. A collection of the images used in the calibration is shown in figure 3.8. The code used for this calibration was adapted from (Lars Tingelstad (2018)).

$$K = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \quad (3.1)$$

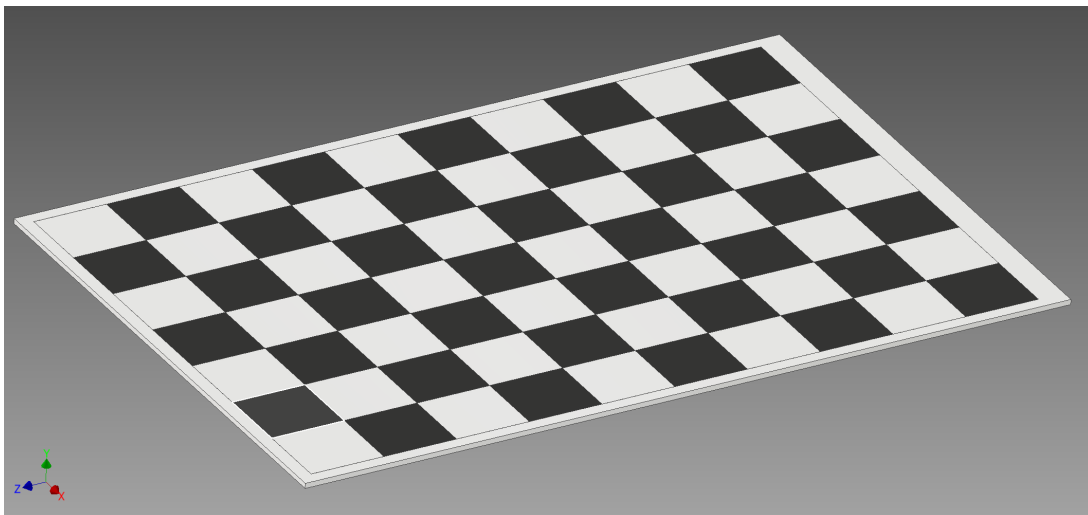


Figure 3.6: 3D-modeled chessboard

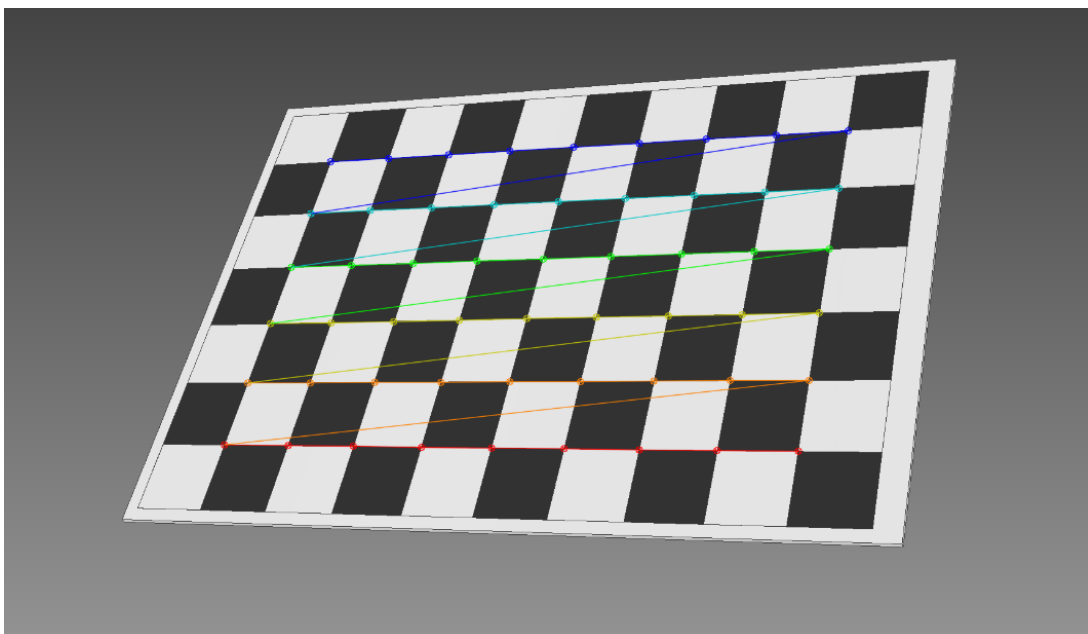


Figure 3.7: 6:9 chess pattern recognized

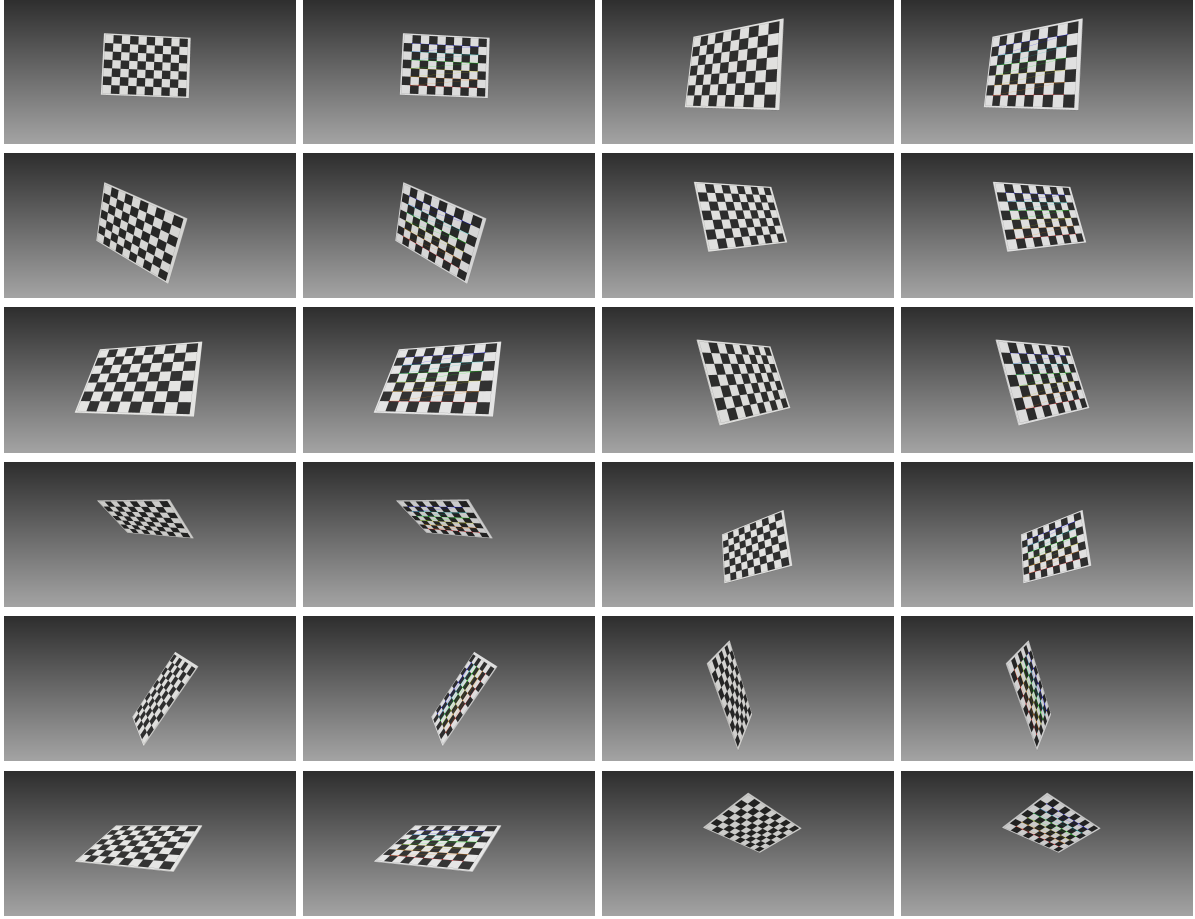


Figure 3.8: Some of the images used for the camera calibration, showing the input image and image after recognizing the chessboards

3.5 Transformations

To get the 3D position of the starting point, some calculations is needed. From the feature detection in the computer vision program, the pixel coordinates can be found, given as:

$$p = \begin{bmatrix} u \\ v \end{bmatrix}, \quad \tilde{p} = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \quad (3.2)$$

Where p is the pixel coordinates and \tilde{p} is the homogeneous pixel coordinates.

To be able to use the pixel coordinates, the relative position of the camera is needed to calculate the position of the weld start. The translation from the camera frame to the world coordinate frame is denoted as t_{cw}^c , where the world frame is located at the face of the chuck, which can be seen from figure 3.5. t_{cw}^c is an input we provide depending on the camera location.

$$t_{cw}^c = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad (3.3)$$

The camera parameter matrix from 3.4 is also needed since it has parameters linking the physical properties of the camera to the image. The camera parameter matrix is given by:

$$K = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix}, \quad f_x = \frac{f}{\rho_w}, \quad f_y = \frac{f}{\rho_h} \quad (3.4)$$

The next step is to find the position from the camera to the weld start, as seen from the camera, denoted r_{cp}^c , see equation 3.5. To find r_{cp}^c the pixel coordinates is rewritten to the same form as the normalized image coordinates, which can be see from equation 3.6. Further the equation 3.8 shows the connection between r_{cp}^c and the now homogeneous normalized image coordinates.

$$r_{cp}^c = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \quad \tilde{r}_{cp}^c = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (3.5)$$

$$s_x = \frac{(u - u_0)}{f_x}, \quad s_y = \frac{(v - v_0)}{f_y} \quad (3.6)$$

$$s = \begin{bmatrix} s_x \\ s_y \end{bmatrix} = \begin{bmatrix} x/z \\ y/z \end{bmatrix} \quad (3.7)$$

$$\tilde{s} = \begin{bmatrix} s_x \\ s_y \\ 1 \end{bmatrix} = \begin{bmatrix} x/z \\ y/z \\ 1 \end{bmatrix} = \frac{1}{z} r_{cp}^c \quad (3.8)$$

To align the camera frame with the same orientation as the world frame, two rotation matrices is necessary, first a rotation $\theta = \frac{pi}{2}$ radians around the x-axis, followed by a rotation $\phi = -\frac{pi}{2}$ around the z-axis of the rotated frame. The combined rotation matrix is given below.

$$R_{cw}^c = \begin{bmatrix} \cos(\phi) & -\sin(\phi) & 0 \\ \cos(\theta)\sin(\phi) & \cos(\theta)\cos(\phi) & -\sin(\theta) \\ -\sin(\theta)\sin(\phi) & \sin(\theta)\cos(\phi) & \cos(\theta) \end{bmatrix} \quad (3.9)$$

The transformation matrix T_{cw}^c connects the pose of the camera frame and the pose of the world frame. T_{cw}^c is further given by:

$$T_{cw}^c = \begin{bmatrix} R_{cw}^c & t_{cw}^c \\ \mathbf{0} & 1 \end{bmatrix} = \begin{bmatrix} \cos(\phi) & -\sin(\phi) & 0 & x \\ \cos(\theta)\sin(\phi) & \cos(\theta)\cos(\phi) & -\sin(\theta) & y \\ -\sin(\theta)\sin(\phi) & \sin(\theta)\cos(\phi) & \cos(\theta) & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (3.10)$$

and finally, \tilde{r}_{wp}^w , the position of the weld start relative to the world frame can be found:

$$\tilde{r}_{wp}^w = T_{cw}^c{}^{-1} \tilde{r}_{cp}^c \quad (3.11)$$

$$\tilde{r}_{wp}^w = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}, \quad r_{wp}^w = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad (3.12)$$

Chapter 4

Results

4.1 Image processing

Results after the image processing, the images used are print screens from inventor as well as an actual drill pipe for the last image.

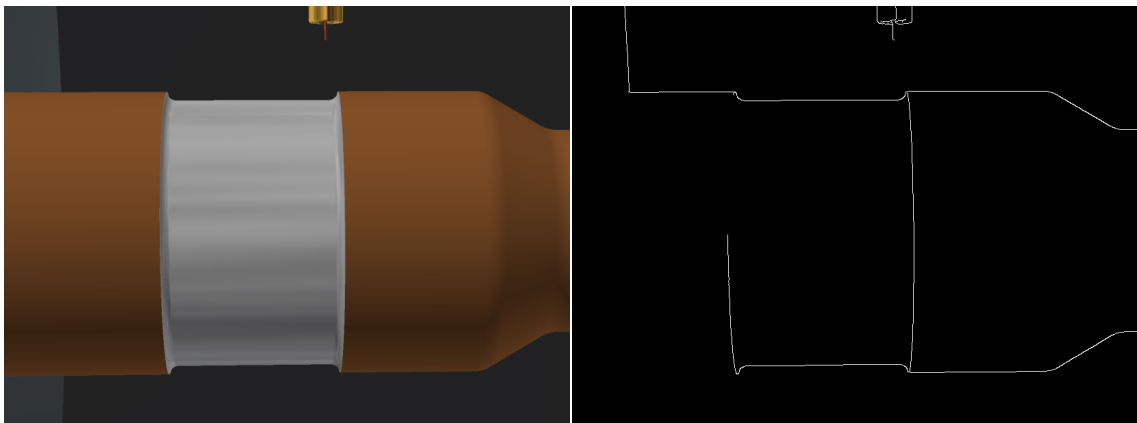


Figure 4.1: Drill pipe with smooth surface

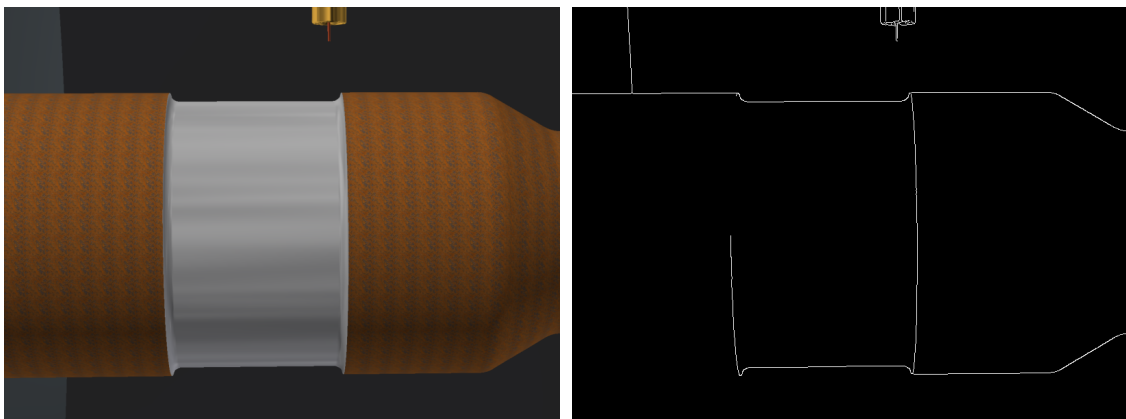


Figure 4.2: Drill pipe with textured surface

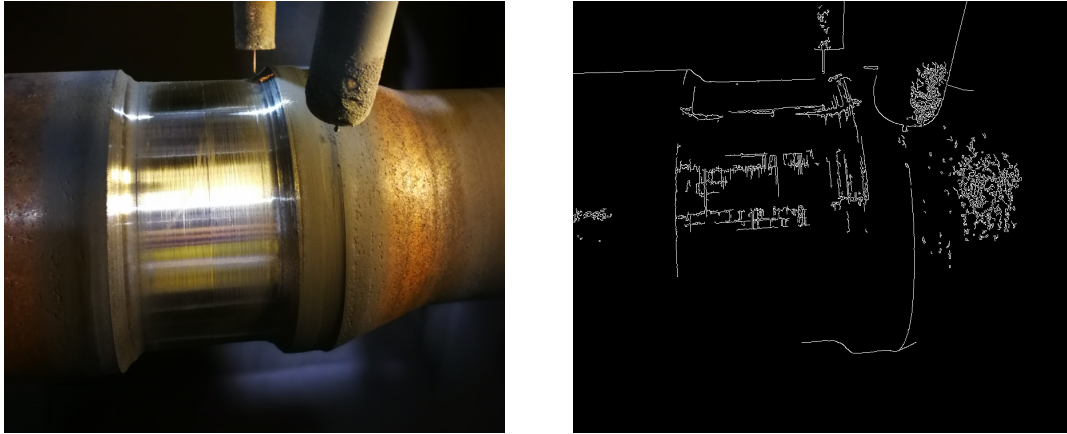


Figure 4.3: Real drill pipe

4.2 Detecting features

The results from taking the results from the edge detection, and running it through the feature detector can be seen below.

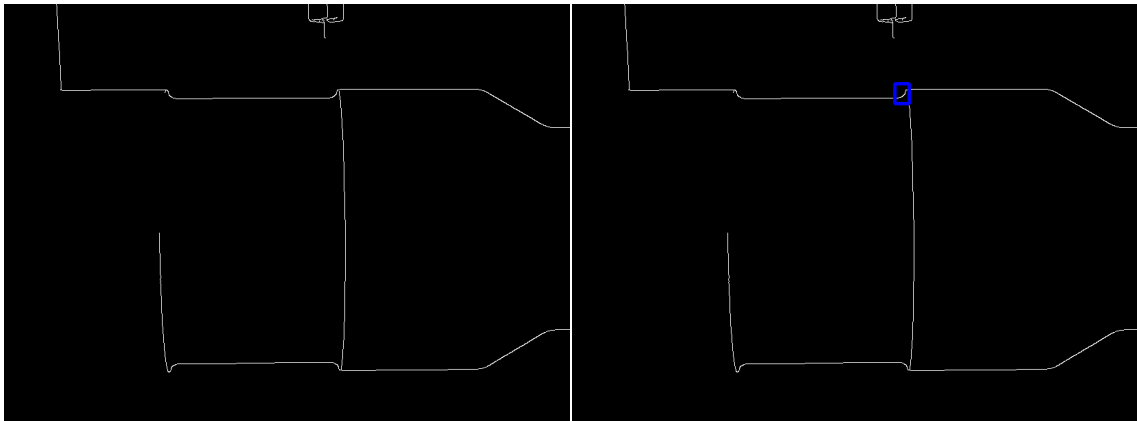


Figure 4.4: Drill pipe with smooth surface, threshold = 0.38

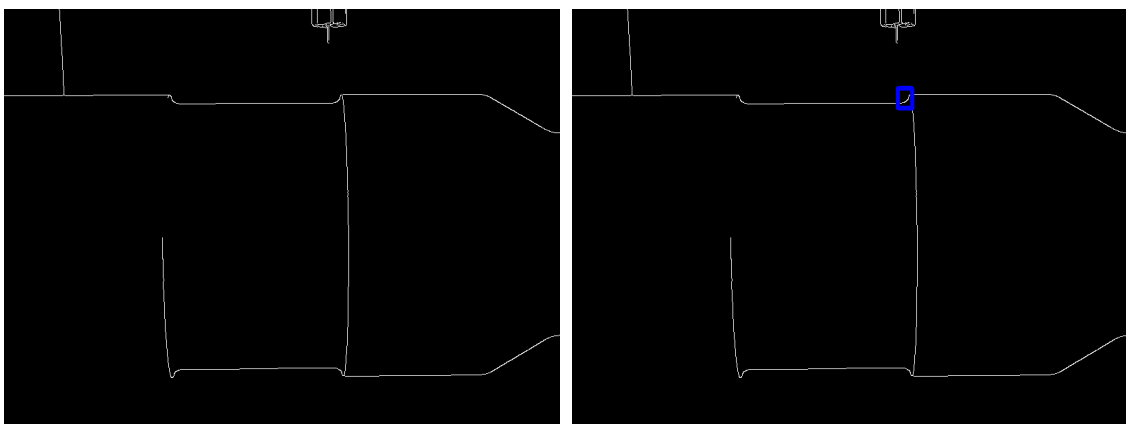


Figure 4.5: Drill pipe with textured surface, threshold = 0.38

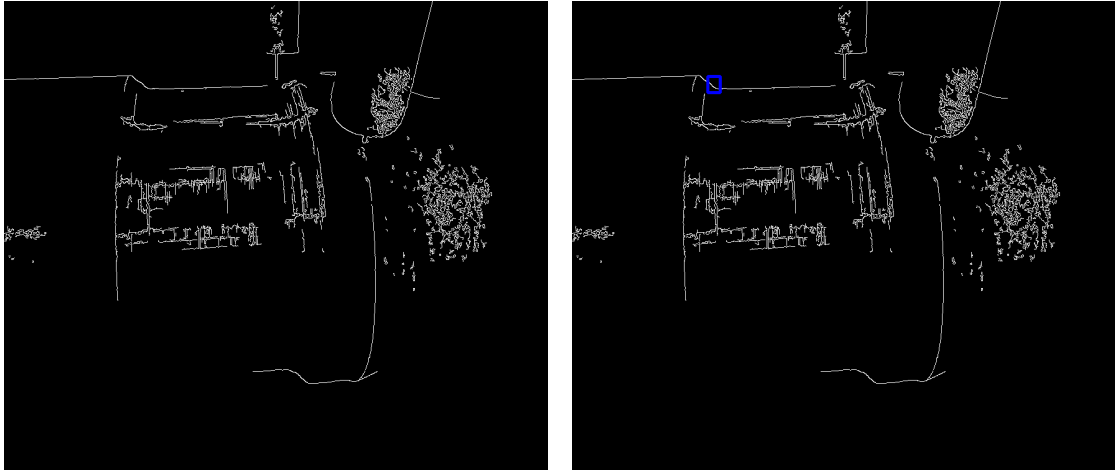


Figure 4.6: Real drill pipe, threshold = 0.45. Note that shadow from the weld gun made the feature detection on the right side of pipe to difficult for the current detector. Therefore another feature detection image was used, which was the other quarter of a circle, which is why the detected weld start is on the other side of the groove.

Figure 4.7 show the detected feature overlaid to the original image.

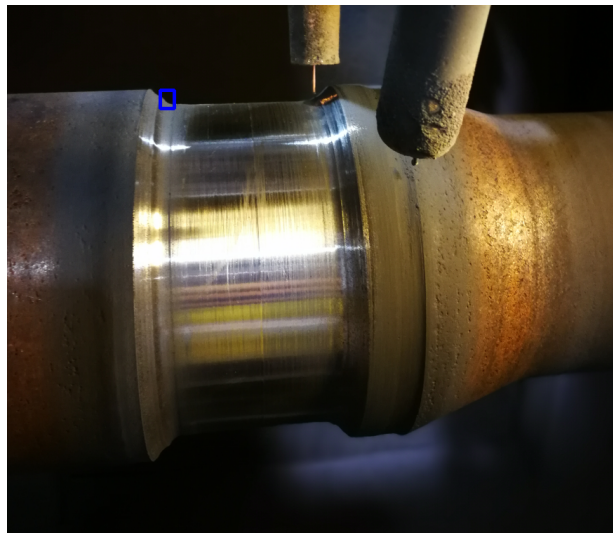


Figure 4.7: Result overlaid to original image

The final results are from an image taken at the known camera location in inventor.

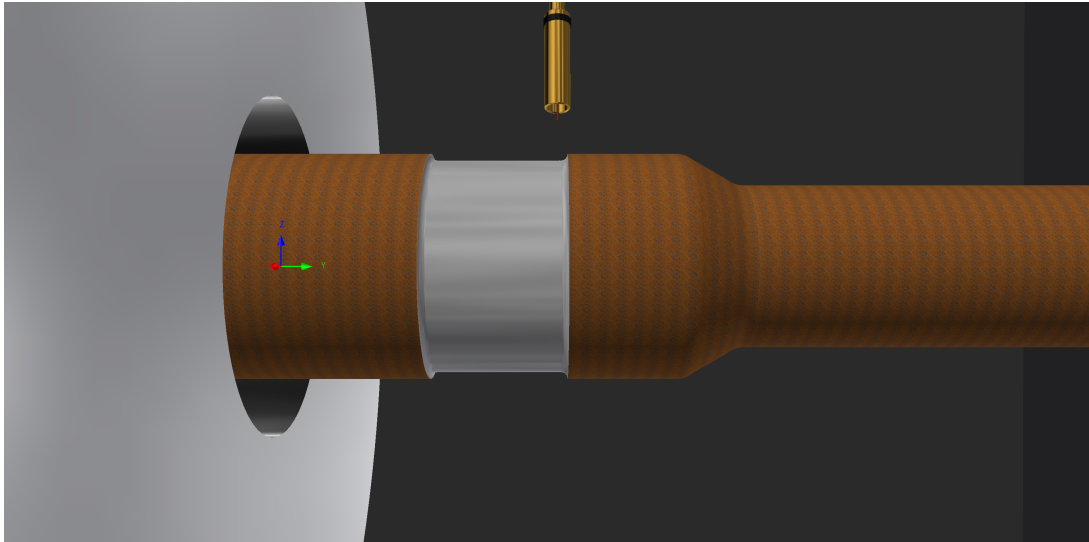


Figure 4.8: Original image from inventor-camera, inventor zoom angle at 60 degrees

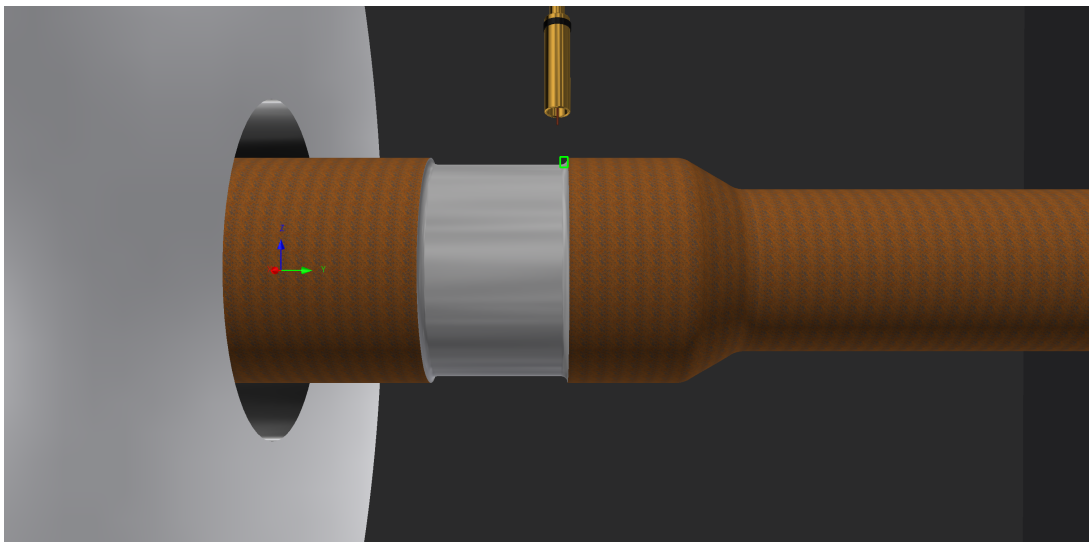


Figure 4.9: Detected start-point for welding, overlayed on original image

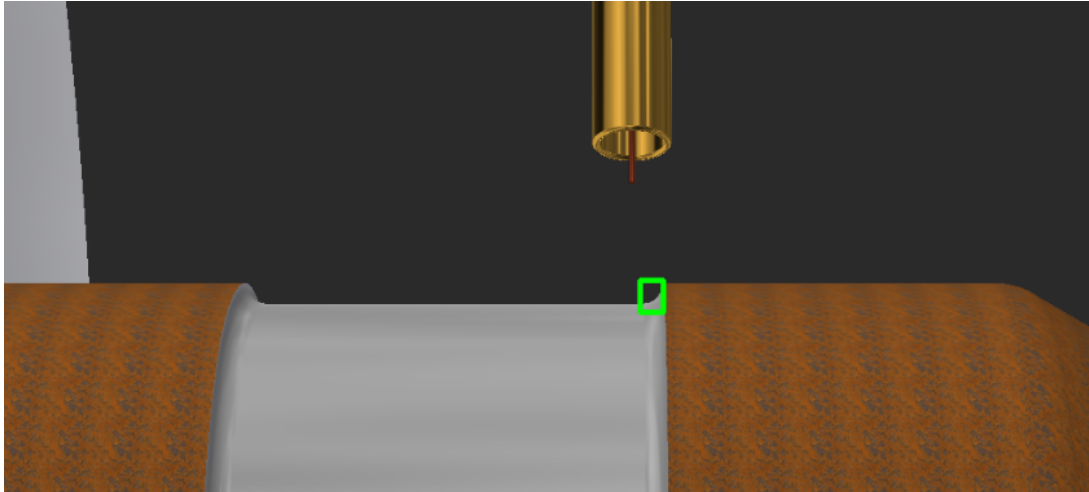


Figure 4.10: zoomed result

The feature detection program located the weld-start at the following pixel coordinate:

$$p = \begin{bmatrix} 1539 \\ 417 \end{bmatrix} \quad (4.1)$$

4.3 Estimated weld start

From Autodesk inventor, all the distances from frames to other frames as well as from frames to points is given below. Note that the w denotes the world frame, p denotes the point of the weld start, while c denotes the camera frame.

$$r_{wp}^w = \begin{bmatrix} 0 \\ 0.207157 \\ 0.078 \end{bmatrix}, \quad r_{cp}^c = \begin{bmatrix} 0.007157 \\ 0.078 \\ 0.7 \end{bmatrix}, \quad r_{cw}^c = \begin{bmatrix} -0.2 \\ 0 \\ 0.7 \end{bmatrix} \quad (4.2)$$

The camera calibration in autodesk inventor gave the following results, note that u_0 and v_0 was placed manually from looking at image size.

$$K = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2609.543 & 0.00 & 1508.5 \\ 0 & 2607.044 & 747 \\ 0 & 0 & 1 \end{bmatrix} \quad (4.3)$$

The camera calibration function also returned the mean Re-projection error, $RMS = 0.5172$

Calculating the steps from the transformation part in section 3.5, yields the following result for the weld start position seen from the world frame:

$$r_{wp}^w = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0.20818 \\ 0.0886 \end{bmatrix} \quad (4.4)$$

The error between the actual position and the estimated position in meters is then given by:

$$r_{wp\ error}^w = \begin{bmatrix} 0 \\ 0.207157 \\ 0.078 \end{bmatrix} - \begin{bmatrix} 0 \\ 0.20818 \\ 0.0886 \end{bmatrix} = \begin{bmatrix} 0 \\ -0.001023 \\ -0.0106 \end{bmatrix} \quad (4.5)$$

Chapter 5

Discussion

5.1 Image processing

From the results in section 4.1, we can see that the current edge detector does not have any problems with the images taken from the 3D-model. The additional disturbances from the texture in figure 4.1 is no problem after some tuning of parameters. When we look at the image of the real drill pipe in figure 4.3, we see some disturbances from the light reflecting as well as the rough texture of the drill pipe. However, some of these disturbances could be addressed by additional tuning, or by changing the actual light setting and camera position in the setup. The image from figure 4.3 is far from an optimal setting, with some trial and error, a better setup might improve the results substantially. It would also be possible to try different edge detection algorithms in combination with different settings to get an optimal result.

5.2 Detecting features

The feature detection was successful for all the images in the paper. The method of looking for a quarter of a circle, is a very simple way to detect the beginning and end of the groove left after the carbide cutter. The method proved rather reliable for the images of the 3D-model, but not for the ideal noises in the real image of the drill pipe seen in figure 4.6, the method will most likely not be as robust. As mentioned in section 5.1, if the edge detection managed to erase more noise, the method might still be a viable option.

It might be a good idea to mount a stationary camera in the existing weld-stations, and take images as the operators work. First, one could take an image of the pipe without the weld gun being close to the weld start, and then take another image just before the operator starts the weld, with the weld gun at the correct position provided by the operator. Then these images could be used to train a neural network to detect the correct weld start and stop. The position of the weld gun would work as the solution to which the neural network checks its own guess. This could also be tried in Autodesk Inventor, where one could generate several images with realistic variations, and use a visible point in Inventor as the correct answer to where the optimal weld start is located.

5.3 Setup

The use of Inventor was critical to be able to test how well the program worked. One of the challenges with the real images of the drill pipes, was that the location from where the image was taken is unknown. As the transformation from pixel coordinates to the 3D coordinates depends on the intrinsic parameters of the camera used, the intrinsic parameters from the Inventor camera needed to be found. The ability to use a camera view in Inventor allowed for a precise setup at the desired location. Arriving to the camera parameter matrix was not done inside the camera function of Inventor, as the precise location from where these images was taken, is not essential for the calibration part. In hindsight, these images

for the calibration should all be taken within the camera function in Inventor due to the fact that the Inventor camera has a different zoom angle than the regular environment in Inventor. When using the camera function, one has to set factors such as the zoom-angle. Since the zoom angle in the regular Inventor environment view was not necessarily the same as the default in the camera function, this could create some errors.

5.4 Transformation

The transformation from the pixel coordinates to the actual 3D coordinates was rather straight forward, as can be seen from section 3.5. When looking at the final result, we see that there was an error between the actual position and the estimated position from the pixel coordinates. This might be the result of the way the camera parameter matrix was derived, as explained in section 5.3. Some tuning of the camera zoom angle was tested, and naturally, it did have an impact on the final error. A solution might be to create the camera parameter matrix one more time, ensuring that the zoom angle is the same for the chessboards images and the test image.

Additionally, the placement of the camera, which was looking at the centre of the pipe, might cause deviations in the estimated height of the weld-start due to the curvature of the pipe. This could be improved by raising the camera higher so that it is more aligned with the height of the weld-start. Another source for the height error is the fact that the pixel coordinates are located at the center of the located feature, which is the center of the green rectangle seen in figure 4.10. While the actual position is located at the minimum diameter of the groove. The sum of the mentioned potential errors might explain the positional error from the final result.

Chapter 6

Conclusion and future work

6.1 Conclusion

This paper is part of a larger project, to create a new production line for handling most of the time consuming process of soft-legging drill pipes. The paper address and test how computer vision could be used as a tool to help automate the process of welding the soft-legging layer on drill pipes. One of the main focuses of the work has been to create feasible solutions, which could be implemented with as little difficulty as possible.

To summarize, in this paper the following has been achieved:

- Create a program for detecting the edges of a drill pipe which can be tuned easily.
- Create a program for detecting features such as detecting the start and stop position of the machined groove for the soft-lagging weld.
- Create a program for transforming pixel coordinates to actual 3D coordinates

Neither of the programs have been optimized, and more work is required before it can be put to use in a production line. However, the programs clearly work as a proof of concept as to how computer vision can be an option for controlling a welding robot. The results are promising, and with some corrections to eliminate the error between the actual position and the estimated position, the use of computer vision in the production might create advantages in both repeatability and optimizing the weld process.

6.2 Future work

Based on the programs created in this paper, additional applications can be added. The location of the weld start and stop, as well as the diameter difference of the groove and tool joint, can be utilized to optimize welding parameters. The start and stop of the weld will indicate the region of interest for the groove. From here, a search for horizontal lines can be used and the groove diameter can be estimated from the pixel coordinates of the two lines.

Computer vision should also be considered to control the lathe machining the grooves. The program should also be able to validate the surface finish of the groove to look for pores and decide whether or not the machined groove is deep enough. The physical design of the system can also be continued, with the insights from the two papers written on the subject. This includes the logistics of the drill pipes on their way to the weld station, the rotation of the drill pipe while welding, and the cooling of the drill pipes to prevent damage to the inner coating.

Bibliography

Alexander Mordvintsev, Abid K. Canny edge detection. web page, december 2013. URL https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_canny/py_canny.html. Last checked: 16.10.18.

Olav Egeland. *A note on robot kinematics*. Unknown, 2018a.

Olav Egeland. *A note on vision*. unknown, 2018b.

Lars Tingelstad. Camera calibration. web page, october 2018. URL https://github.com/tingelst/tpk4170-robotics/blob/master/tpk4170/camera_calibration/CameraCalibration.ipynb. Last checked: 30.11.18.

Espen Liavik. *Utilizing laser triangulation to extract spatial coordinates of drill-pipe's geometry*. unknown, 2018.

OpenCV team. Smoothing images. web page, december 2015. URL https://docs.opencv.org/3.1.0/d4/d13/tutorial_py_filtering.html. Last checked: 16.10.18.

OpenCV team. About opencv. web page, November 2018. URL <https://opencv.org/about.html>. Last checked: 23.11.18.

Appendices

Appendix A

Python code

A.1 Main code

```
1 # -*- coding: utf-8 -*-
2 """
3 @author: Kristoffer Hermansen
4 """
5
6 import cv2
7 import sys
8
9 # Read image
10 path = "C:\\Users\\Kristoffer\\Documents\\NINU\\Prosjektoppgave\\TTK4551
    Specialization Project\\Code\\Main code\\"
11 filename = "dp_texture.png" #Filename Input
12 savefile = "dp_texture_save.png" #Savename image with edges
13 detectFile = "detect_start.png" # Feature to detect
14 saveresult = "dp_texture_result.png"#Savename image detected feature
15 threshold = 0.35
16 image = path + filename
17
18 img = cv2.imread(image,0) # image is grey scale due to (,0)
19 img1 = cv2.imread(image,1) # used to overlay result on original image
20
21 # Creating Trackbars, mostly usefull for tuning new images
22 def nothing(x):
23     pass
24 cv2.namedWindow('blured')
25 cv2.createTrackbar('Sigma','blured',1,50,nothing)
26 cv2.createTrackbar('Kernel','blured',1,50,nothing)
27
28 cv2.namedWindow('canny')
29 cv2.createTrackbar('canny1','canny',100,300,nothing)
30 cv2.createTrackbar('canny2','canny',250,300,nothing)
31 print('start tuning, press esc to continue')
32 while True: # while loop which allows for tuning using trackbar
33     # assign parameter sigma to the trackbar in bluring window
34     sigma = cv2.getTrackbarPos('Sigma','blured')
35     # preventing sigma from becoming 0 and crash program
36     sigma = sigma+1
37     # assign parameter kernel to the trackbar in bluring window
38     kernel = cv2.getTrackbarPos('Kernel','blured')
39     # making sure kernel is always odd
40     kernel = 2*kernel-1
41
42     # assigning parameter canny1 to trackbar in canny window
```

```

43 canny1 = cv2.getTrackbarPos('canny1','canny')
44 # assigning parameter canny2 to trackbar in canny window
45 canny2 = cv2.getTrackbarPos('canny2','canny')
46
47 # applying gaussian filter to the image
48 GaussianBlur = cv2.GaussianBlur(img,(kernel,kernel),sigma)
49 # applying canny edge detection to the blurred image
50 canny = cv2.Canny(GaussianBlur,canny1,canny2)
51
52 # Showing images
53 cv2.imshow('image',img)
54 cv2.imshow('blured',GaussianBlur)
55 cv2.imshow('canny',canny)
56
57 # While loop stopper
58 key = cv2.waitKey(1)
59 if key == 27:
60     print('while loop stpped')
61     break
62 # waiting for user to inspect images, press esc to break while loop
63 print('Press "esc" to close program or press "s" to continue')
64 k = cv2.waitKey(0) & 0xFF
65
66 # Terminate programme
67 if k == 27: # wait for ESC key to exit
68     print('Program terminated by user! ')
69     cv2.destroyAllWindows()
70     sys.exit()
71 elif k == ord('s'): # wait for 's' key to save and exit
72     cv2.imwrite(savefile,canny)
73     print('Edge detected image saved')
74     cv2.destroyAllWindows()
75
76 ###
77
78 # Detecting the start of weld
79 import numpy as np
80 def detector(search,detect,threshold,color):
81     # Import image to search in
82     print('Searching in:',search)
83     print('Looking for:',detect)
84
85     image = path + search
86     # Import feature to search for
87     image_search = path + detect
88
89
90     img_bgr = cv2.imread(image)
91     img_gray = cv2.cvtColor(img_bgr,cv2.COLOR_BGR2GRAY)
92
93     X_0 = cv2.imread(image_search,0)
94     X_0_Blur = cv2.GaussianBlur(X_0,(1,1),2)
95     # applying canny edge detection to the blurred image
96     canny = cv2.Canny(X_0_Blur,100,200)
97     X_0 = canny
98     # Get the size of image
99     w, h = X_0.shape[: -1]
100     w1, h1 = img_gray.shape[: -1]
101     print('Image width in pixels',w1)
102     print('Image height in pixels',h1)
103
104     res = cv2.matchTemplate(img_gray,X_0,cv2.TM_CCOEFF_NORMED)
105     location = np.where(res>= threshold)

```

```

106     # print('The detected feature is located at pixels: ',location)
107
108
109     # Note the if statment ignoring false points, limits depend on image
110     u_lim = w1/2
111     v_lim = h1/2
112     for point in zip(*location[:, :-1]):
113         if point[1] > v_lim or point[0] < u_lim: # choses which points to ignore
114             pass
115         else:
116             print('Feature detected at pixel coordinates: ')
117             print('u = ',point[0])
118             print('v = ',point[1])
119             cv2.rectangle(img1,point,(point[0]+w, point[1]+h),(color),3)
120 #             img_bgr = cv2.resize(img_bgr,(1280,900))
121             cv2.imshow('detected',img1)
122             cv2.imshow('image search',X_0)
123             cv2.imwrite(saveresult,img1)
124             print('Image saved')
125 #             cv2.waitKey(0)
126             # While loop stopper
127         key = cv2.waitKey(1)
128         if key == 27:
129             print('while loop engaged')
130 #             break
131             cv2.destroyAllWindows()
132
133
134 detector(savefile,detectFile,threshold,(0,255,0))
135
136 cv2.waitKey(0)
137 cv2.destroyAllWindows()

```


A.2 Transformation

```

1  # -*- coding: utf-8 -*-
2  """
3  @author: Kristoffer Hermansen
4  """
5
6  import numpy as np
7  pi = np.pi
8  def rotx(theta):
9      ct = np.cos(theta);
10     st = np.sin(theta);
11     R = np.array([[1, 0, 0],
12                  [0, ct, -st],
13                  [0, st, ct]])
14     return R # theta rotation about x-axis
15  def roty(theta):
16     ct = np.cos(theta);
17     st = np.sin(theta);
18     R = np.array([[ct, 0, st],
19                  [0, 1, 0],
20                  [-st, 0, ct]])
21     return R # theta rotation about y-axis
22  def rotz(theta):
23     ct = np.cos(theta);
24     st = np.sin(theta);
25     R = np.array([[ct, -st, 0],
26                  [st, ct, 0],
27                  [0, 0, 1]])
28     return R # theta rotation about z-axis
29  def tran(R,v):
30     T = np.array([[R[0][0], R[0][1], R[0][2], v[0]],
31                  [R[1][0], R[1][1], R[1][2], v[1]],
32                  [R[2][0], R[2][1], R[2][2], v[2]],
33                  [0, 0, 0, 1]])
34     return np.round(T,3)
35
36  # Insert K matrix from calibration
37  K = np.array([[2609.543, 0, 3017/2],
38               [0, 2607.044, 1494/2],
39               [0, 0, 1]])
40
41  empt = []
42  # z is distance between camera and point along optical axis
43  z = 0.7
44  # Pixel coordinates derived from Main.code:
45  p = np.array([[1539],
46               [417]])
47  tc_co = np.array([[ -0.2],
48                  [0],
49                  [0.7]])
50
51  print('Given pixel coordinates from computer vision: \n',p)
52  s_x = (p[0]-K[0][2])/K[0][0]
53  s_y = (p[1]-K[1][2])/K[1][1]
54  s = np.append(empt,[s_x,s_y])
55  s = np.reshape(s,(2,1))
56  print('\n Normalized image coordinates: \n',s)
57  s_tilde = np.append(s,1)
58  s_tilde = np.reshape(s_tilde,(3,1))
59  rc_cp = (s_tilde)*z
60  print('\n Start positon from camera frame: \n',rc_cp)
61  rc_cp_tilde = np.append(rc_cp,1)

```

```
62 rc_cp_tilde = np.reshape(rc_cp_tilde,(4,1))
63 Rx = rotx(pi/2)
64 Rz = rotz(-pi/2)
65 RT = Rx@Rz
66 T = tran(RT,tc_co)
67 T_inv = np.linalg.inv(T)
68 ro_op_tilde = T_inv@rc_cp_tilde
69 print('\n Start positon from world frame: \n',ro_op_tilde)
```

A.3 Camera calibration

```

1  # -*- coding: utf-8 -*-
2  """
3  Created on Thu Nov 22 06:14:16 2018
4  Original author: https://github.com/tingelst/tpk4170-robotics/blob/master/tpk4170/
   camera_calibration/CameraCalibration.ipynb
5  Edited by: Kristoffer
6  """
7
8  ##### Camera Calibrateion #####
9  import os
10 from glob import glob
11 import numpy as np
12 np.set_printoptions(suppress=True)
13 import cv2
14 import matplotlib as mpl
15
16 mpl.rcParams['figure.dpi'] = 150 # setter oppl sningen til 150
17
18 def splitfn(fn):
19     path, fn = os.path.split(fn)
20     name, ext = os.path.splitext(fn)
21     return path, name, ext
22
23 img_mask = "C:\\Users\\Kristoffer\\Documents\\NINU\\Prosjektoppgave\\TTK451
   Specialization Project\\Code\\Calibration\\Chess_0?.png"
24 img_names = glob(img_mask)
25 print(img_names)
26
27 #28.50 is chessboard rectangle width
28 # square_size = 1
29 square_size = 0.0285
30 pattern_size = (9, 6)
31 pattern_points = np.zeros((np.prod(pattern_size), 3), np.float32)
32 pattern_points[:, :2] = np.indices(pattern_size).T.reshape(-1, 2)
33 pattern_points *= square_size
34 print(pattern_points[34:40])
35
36 def process_image(fn, debug=False):
37
38     # Read image from file
39     img = cv2.imread(fn, 0)
40     if img is None:
41         print("Failed to load", fn)
42         return None
43
44     # Locate chessboard corners in images
45     found, corners = cv2.findChessboardCorners(img, pattern_size)
46     if found:
47         term = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_COUNT, 30, 0.001)
48         cv2.cornerSubPix(img, corners, (5, 5), (-1, -1), term)
49
50     # Debug: Draw chessboard on image
51     if debug:
52         vis = cv2.cvtColor(img, cv2.COLOR_GRAY2BGR)
53         cv2.drawChessboardCorners(vis, pattern_size, corners, found)
54         _path, name, _ext = splitfn(fn)
55         outfile = os.path.join(_path, name + '_chess.png')
56         cv2.imwrite(outfile, vis)
57
58     # Return None if the chessboard is not found
59     if not found:

```

```

60     print('Chessboard not found')
61     return None
62
63     # Print status
64     print(' {}... OK'.format(fn))
65
66     return (corners.reshape(-1, 2), pattern_points)
67
68 chessboards = [process_image(fn, debug=True) for fn in img_names]
69 chessboards = [x for x in chessboards if x is not None]
70
71
72 # Split image points and objects points
73 img_points = []
74 obj_points = []
75 for (corners, pattern_points) in chessboards:
76     img_points.append(corners)
77     obj_points.append(pattern_points)
78
79 # Find the size of the image
80 img_size = cv2.imread(img_names[0], 0).shape[:2]
81 print('Image size: \n',img_size)
82
83 rms, camera_matrix, dist_coeffs, rvecs, tvecs = cv2.calibrateCamera(obj_points,
84     img_points, img_size, None, None)
85 print('RMS: \n',rms)
86 print('Camera parameter matrix: \n',camera_matrix)

```