NTNU
Norwegian University of
Science and Technology

Advanced Camera Detection and Measurement System in the Czochralski Process

Endre Arnesen Gjølstad

Master of Science in Cybernetics and Robotics
Submission date: June 2018
Supervisor: Morten Hovd, ITK
Co-supervisor: John Atle Bones, SINTEF

Norwegian University of Science and Technology
Department of Engineering Cybernetics

”I have approximate knowledge of many things.”
-Demon Cat
Adventure Time, season 1 episode 18

Sammendrag

I denne oppgaven presenteres Czochralski-prosessen, samt det teknologiske oppsettet som brukes i prosessen. De tre problemene denne oppgaven tar sikte på å løse, presenteres: implementering av automatisk gjenkjenning av kalde krystaller, automatisk gjenkjenning av strukturtap i *body* fasen og automatisk temperaturdetektering i *neck*-stabiliseringsfasen. Etter introduksjonen presenteres den relevante teorien. Denne avhandlingen refererer til den automatiske kalde krystall-deteksjonalgoritmen som er utviklet i et tidligere prosjekt. Dette arbeidet presenteres i et eget kapittel. Deretter beskrives hvert av de tre hovedproblemene og løsninger foreslås, testes og diskuteres for hver av dem separat. På slutten av denne oppgaven er et kapittel med konklusjoner av de foreslåtte problemløsningene. Avhandlingen resulterte i at problemer ble avdekket i den implementerte automatiske kalde krystalldeteksjonen som må analyseres videre og avstemmes for en tilfredsstillende ytelse på fabrikken. For strukturtapgjenkjenning ble flere tilnærminger foreslått. De fleste tilnærmingene kan forkastes, men gjennomsnittsverdi tilnærmingen og maskininnlæringsmetoden virker som gode metoder for en robust gjenkjenning av strukturtapfenomenet. I tilfellet av automatisk temperatur deteksjon i *neck* stabiliseringsfasen, var det problemer med at kameraet ikke hadde en høy nok oppløsning for å få presise resultater. Hvis et nytt kamera er anskaffet, foreslår denne avhandlingen to separate metoder som kan være gunstige for automatisk å oppdage ønsket temperatur i denne fasen. En maskininnlæringsmetode og en dynamisk programmeringsmetode hvor problemet blir delt inn i mindre separate problemer.

Summary

In this thesis the Czochralski process is introduced as well as the technological setup used in the process. The three problems this thesis aim to solve are presented: implementation of automatic cold ingot detection, automatic detection of structure loss in the body phase and automatic temperature detection in the neck stabilization phase. After the introduction, the relevant theory is presented. This thesis references the automatic cold ingot detection algorithm developed in a previous project heavily, so this work is presented in a separate chapter. Then each of the three main problems are described and solutions are proposed, tested and discussed for each of them separately. At the end of this thesis is a chapter with conclusions of the proposed problem solutions. The thesis resulted in problems being uncovered in the implemented automatic cold ingot detection which needs to be further analyzed and tuned for a satisfactory performance in the factory. For the structure loss detection, several approaches were proposed. Most of the approaches can be discarded, but the mean value approach and the machine learning approach both seem viable for a robust detection of the structure loss phenomenon. In the case of automatic temperature detection in the neck stabilization phase, there were problems with the camera not having a high enough resolution to get precise results. If a new camera is acquired this thesis proposes two separate methods which can be viable for automatically detecting the desired temperature in this phase, a machine learning approach and a dynamic programming approach, where the problem is divided into smaller parts.

Preface

Industrial background

Monocrystalline silicon

Monocrystalline silicon is a specific type of silicon crystal in which the atoms are all arranged in a repeating pattern throughout the entire crystal mass. These monocrystalline silicon crystals are desired in the electronic and renewable energy market since they act as highly efficient semiconductors due to their atom arrangement. NorSun AS is Norway's largest producer of monocrystaline silicon and their factory in Årdal encompasses the entire process from growing crystals to cutting them into thin slices known as *wafers*. The wafers are sold to companies producing solar cells throughout the world. Due to new technology increasing energy output and policies resolving to increase the use of renewable energy in the modern world, the demand for monocrystalline silicon is enormous and constantly growing. The current demand is higher than NorSun AS can supply. To meet the increased demand, NorSun AS is always looking to increase their productivity and has hired a team from Sintef to research how to improve the efficiency of the factory.

Present operation

The monocrystalline crystals are grown in a process called the Czochralski process. In this process a single crystal of silicon, commonly called an *ingot*, is produced by dipping a silicon seed into a furnace containing pure melted silicon. NorSun's factory in Årdal has 70 of these furnaces which are called *pullers*. The pullers are highly automated by control systems which are governing the temperature of the melt, the speed at which the ingot is pulled and limiting the diameter growth of the crystal. But the Czochralski process is a very complex process where many small factors and variations can have an effect on the succes rate of growing the monocrystalline silicon ingot. Defects and unwanted states often occur. Today these defects and unwanted states are mainly detected by a visual inspection performed by factory operators at an interval of 30 minutes. These inspections are often as simple as looking into the furnace at the crystal and seeing that there are no defects. When something goes wrong there is no guarantee that the unwanted state or defect will be seen in time to recover the crystal to a desired state. This often results in whole crystals being discarded. In **Chap. 1** the pulling process and the technical setup will be further explored.

Previous work

The previous work was done in a project the fall of 2017. This project aimed to design a computer vision algorithm to detect a phenomenon where the ingots become cold and lose their cylindrical forms. Ultimately a cold ingot can become cold enough to lose its monocrystalline structure due to thermal stress and thus has to be discarded. The following paragraph is the summary of the previous project:

In this project the Czochralski process and the technological setup used to optimize the process was introduced. The problem of detecting cold ingots was presented and broken into four sub-objectives: forming hypotheses, analyzing available tools, designing the algorithms and verifying their performance. Some computer vision theory was introduced and the software framework used was analyzed for potential functionality that is useful for the task. After the theory was introduced two main hypotheses to detect the cold ingot was proposed. Both methods were then described, implemented and tested separately. In the end their results were discussed and compared to each other. The project resulted in an algorithm which was able to detect the cold ingot in all three data sets which it was tested upon.

The previous work is very relevant for the work done in this thesis and will be referenced often. This work will be introduced in greater detail in **Chap. 3**.

Problem Description

The aim of this thesis is to research how some unwanted states and phenomena can be automatically detected by a computer vision approach, thus contributing to an *Advanced camera detection and measurement system for the Czochralski process*. More specifically, the problems this thesis seek to solve are:

- Implementing an automatic cold ingot detection algorithm from the previous work done by the author in the factory and verify its robustness and performance in a live environment.

- Conceptualizing, designing and comparing several approaches for detecting structure loss in an ingot in the body phase.

- Conceptualizing, designing and comparing several approaches for automatically detecting a desired and stable temperature during the neck stabilization phase.

The problems to be explored in this thesis are further introduced in **Sec. 1.4**.

Facilities, data access, tools and support

Tools

Some tools used for this thesis are a requirement to interface easily with the factory setup, while some tools are optional and used simply for predefined functionality.

The requirement for this thesis is that any detection algorithm needs to run on the *Scorpion Vision* framework which is a third party licensed software created by *Tordivel AS*. The other tools used for this thesis are mainly the software libraries listed below.

- Scipy

- Matplotlib
- Numpy
- TensorFlow
- Keras

These software libraries contains different functionality which is useful for optimizing script run time, heavy mathematical algorithms or simply ease of use functionality to decrease implementation time. The libraries are introduced in **Sec. 2.3**.

Test setup

The test setup consist of remotely connecting to a computer which runs the same computer vision software, *Scorpion Vision*, as the live factory setup. Any new implementations or tests of new algorithms are implemented and analyzed on the test setup.

Data access

The data which is analyzed on the test setup are sets of images captured on the live setup. The image sets used for testing is captured at a rate of one picture every 20 seconds. The image sets are acquired through requesting images of specific phenomena to someone with access to Sintef's puller in the factory. Then a recorded run containing the specific phenomena has to be identified before extracting the captured image set.

For real time logging in the factory a third party system called *APIS* is used which is hosted by a company called *Prediktor AS*. To receive any logged data from the factory, a request has to be made to either Co-Supervisor John Atle Bones or process engineers at the factory.

The real time logging in the factory and the data received from logging the same ingot on the test setup may vary. This is because the factory setup evaluates five images every second compared to the rate of one image every 20 seconds in the captured data sets which are evaluated on the test setup.

Support

The work done in this thesis is done for a Sintef research team whom are hired by NorSun AS to help modernizing the silicon crystal growth process in their factory. The researchers have their own crystal puller setup at the factory with their own control systems, computer vision software and hardware. Most of my work has been focused on designing proof of concept computer vision algorithms to automate the detection of unwanted states in the crystal growth process. For the design of algorithms I have been working on the test setup at NTNU with image sets collected from the factory.

For the most part this work has been done quite independently with only general guidelines on which phenomena to detect, from my co-supervisor John Atle Bones who is a part of the Sintef research team. He has also helped me understand the characteristics of the Czochralski process and given tips on possible ways of approaching some of the detection problems.

For gathering data, discussing trends seen in the algorithm output and planning how the factory operators should handle new alarms as a result of my detection algorithms I have been in contact with process engineer Jeroen Van Delft and discipline leader Helge Hovland in the NorSun AS factory.

Table of Contents

List of Tables

List of Figures

Abbreviations

Symbol	=	definition
Cz	=	Czochralski as in the Czochralski process
ROI	=	Region of interest
SDev	=	Standard Deviation
OPC	=	Open Platform Communications
PLC	=	Programmable Logic Controller
GUI	=	Graphical User Interface
px	=	Pixel
FIFO	=	First In First Out

Chapter 1

Introduction

The renewable energy market is rapidly expanding with new technology increasing energy output and policies resolving to increase the use of renewable energy throughout the modern world. Due to this rapid market expansion, many companies are working hard to earn their share of the market and the competition is fierce. To stay on top of this competition, the efficiency of the renewable energy sources need to be high quality while the production of the sources need to be as cheap as possible. The growth of ingots is done through the Czochralski process. It is a complex process with many factors affecting the end result and thus there is continuously work being done on new ways to optimize the efficiency of this process. Researchers from Sintef have been hired to automate larger parts of the process. An important step in the automation of the Czochralski process is utilizing a camera and computer vision algorithms to detect states and perform measurements in real time during the process. The thesis work will be divided in three parts. Part one is focused on implementing and testing a previously designed algorithm in the factory. Part two and three are focused on designing new proof-of-concept algorithms for detection of two other phenomena.

1.1 Background

The Czochralski process is a process in which a cylindrical single crystal of silicon is produced by dipping a silicon seed into a furnace containing pure melted silicon. The seed is pulled slowly upwards while being rotated. The melted silicon will attach to the seed so the pulled silicon cools down. As the silicon cools down it will solidify, resulting in a monocrystalline silicon ingot. There are several phases the crystal advances through within the furnace to get the desired properties such as length and diameter while retaining its monocrystalline structure. The phases are visualized in **Fig. 1.1**. The size of the ingot is grown in the body phase of the process and a succesful body phase is therefore important for the commercial result of the process. When the Czochralski process is finished, the monocrystalline silicon ingot is extracted and cut into thin slices known as *wafers* which are further treated. The wafers are typically used either in solar cells or as a foundation

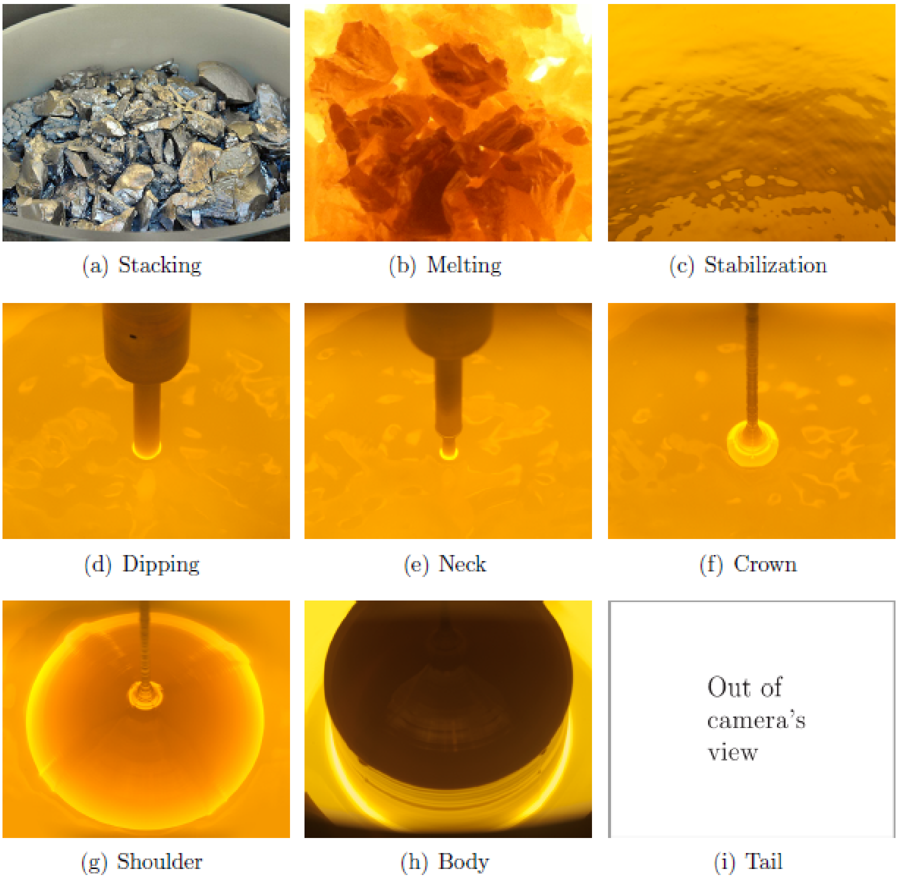

(a) Stacking
(b) Melting
(c) Stabilization
(d) Dipping
(e) Neck
(f) Crown
(g) Shoulder
(h) Body
(i) Tail

Figure 1.1: Main phases of the Czochralski process. Bones (2012)

upon which microelectronic components are deposited.

Stacking as seen in **Fig. 1.1a** is the process where the raw silicon is placed in a crucible which is placed inside the furnace. How the silicon is stacked has an influence on how succesful the growth process will be and is under constant research for improvement. The next step is Melting as seen in **Fig. 1.1b**. In the melting phase, the crucible containing the stacked silicon is placed in the furnace and the raw silicon is melted to a liquid silicon melt. Once all the silicon has melted and becomes a liquid, the melt is often volatile due to cyclical fluctuations and difference in temperature. The stabilization phase from **Fig. 1.1c** is a process of bringing the melt to a stable temperature which reduces its volatility. When the melt is stable an operator will manually adjust the crucible position and start the dipping phase seen in **Fig. 1.1d**. This is where the silicon seed is lowered into the melt to attach the liquid silicon to the seed. After the dipping the growth of a neck is performed in the neck phase shown in **Fig. 1.1e**. The Neck serves as a connection between the pulling mechanism and the ingot which can be easily cut and handled when the ingot is extracted from the puller. When the seed is dipped into the melt the silicon will immediately experience dislocations. Growing the neck helps distance the crystal growth from these initial dislocations. When the neck is grown to a satisying length, another temperature stabilization is performed which is important to ensure good initial conditions for the rest of the crystal growth. When the neck temperature stabilization is completed the crown phase begins. In this phase the pull speed is adjusted to allow to formation of the crown as seen in **Fig. 1.1f**. The structural form of the crystal is formed in the crown which will extend to the rest of the crystal. If the crown has a good structure the shoulder seen in **Fig. 1.1g** will let the ingot grow outwards to the desired diameter. Once the ingot has the desired diameter the body phase seen in **Fig. 1.1h** can be started. This phase adjusts the pull speed and temperature continuously to keep the desired diameter while growing the length of the ingot. When the ingot growth is completed the pull speed increases to close the ingot. This is accomplished by reducing the diameter by increasing the pull speed which gives the ingot a cone form at the bottom. This phase is called the tail phase and is shown in fig **Fig. 1.1i**.

1.2 The Czochralski puller

The Czochralski puller is a big furnace as seen in **Fig. 1.2**. It is specifically created to melt silicon and pull crystal ingots. It consists of a pulling mechanism at the top, space for the crucible which contains the silicon melt, heaters at both sides of the crucible. A lift to adjust the crucible height position as well as heat shielded windows to allow observation of the process, either by visual inspection or by cameras. It is also well insulated to avoid heat radiation. The factory in Årdal as seen in **Fig. 1.3** has 70 of these pullers operating in a large hall.

1.3 Technological setup of the furnace

To consistently produce ingots with desired properties and quality there are automated control systems installed on the furnaces in the factory, implemented as seen in Figure 1.4. These furnaces are equipped with sensors, cameras and control systems. They are commonly referred to as *pullers*.

The pullers are conventionally automated by three controllers

- Automatic diameter controller (ADC),
- Automatic temperature controller (ATC),
- Automatic growth rate controller (AGC).

During the body phase the melt level decreases as the crystal grows in size. This dynamic results in less heat transferred to the melt due to a decrease in surface area in contact with the heaters. The *ATC* follows an empirically determined setpoint trajectory during this phase to remedy heat loss in the melt. Due to small variations in pullers the set point

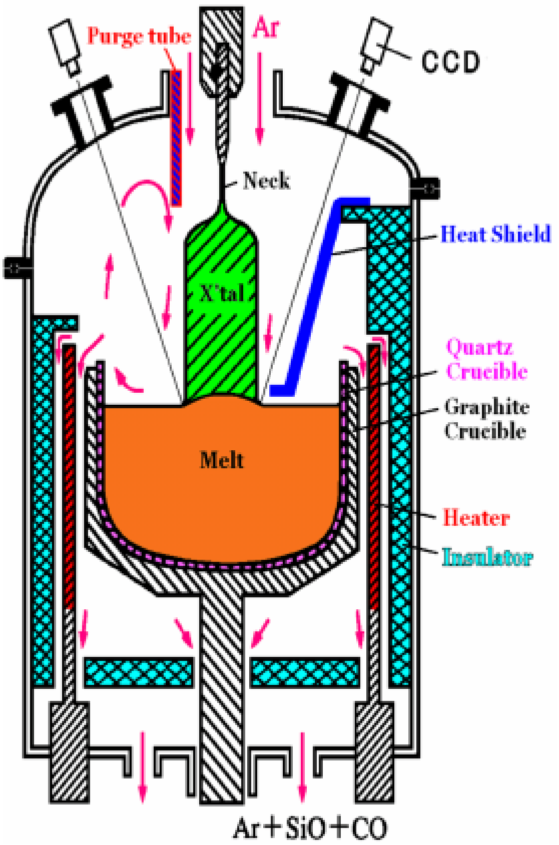


Figure 1.2: A visualization of a Czochralski puller. Kakimoto (2013)

Figure 1.3: The NorSun factory's Czochralski pulling area. NorSun (2018)

will not necessarily result in an optimal temperature for every puller. To minimize the potential error in temperature the *AGC* will add a temperature offset to the setpoint when the pulling rate output by the *ADC* differs from its setpoint. The setpoint trajectory of the *ATC* is tuned to maximize the speed of pulling without losing the monocrystalline structure or the desired diameter.Lee et al. (2005) The *ADC* uses a camera mounted on the puller to measure the current diameter of the ingot. The software used to calculate the measurement on the pullers is called *Scorpion*. It is a computer vision framework with source code access for continued addition or modification of functionality.

1.4 Problem description

NorSun AS is Norway's largest producer of monocrystalline silicon and the demand for their silicon is bigger than what they can supply. Because of the huge demand, NorSun AS are looking to maximize their factory efficiency. An important step in increasing factory efficiency is automating control and measurement operations.

The most critical variable for successful ingot growth is the temperature of the silicon melt and the temperature of the growing silicon ingot. The desired temperature changes slightly depending on individual varying factors within each furnace. It is also difficult to measure the precise temperature of the melt due to the circulation of liquid silicon, which results in cyclical temperature fluctuations. Instead of directly measuring the heat,

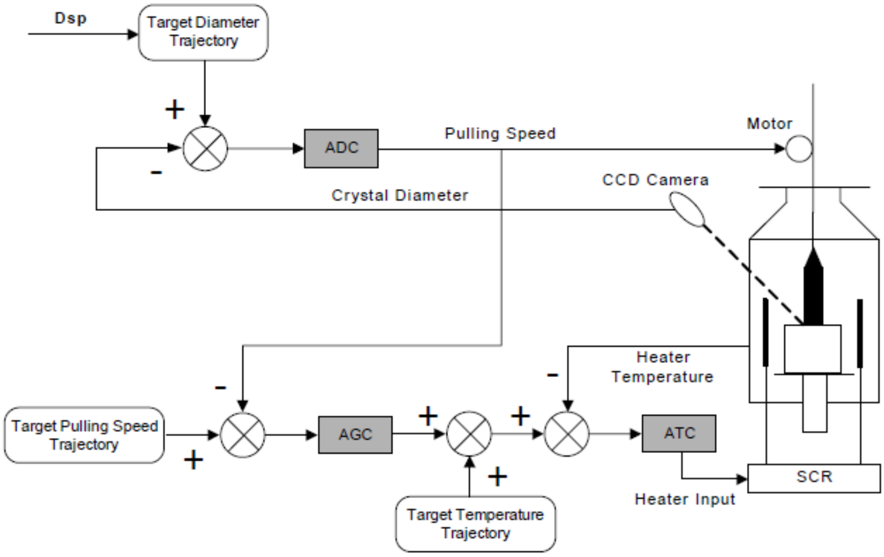


Figure 1.4: Visualization of the control system on the pullers. Lee et al. (2005)

the desired and undesired states of the ingot is detectable through geometrical changes of the ingot. By automatic detection of the phenomena through computer vision techniques, quick discovery of unwanted states with a high level of precision is achievable compared to the sporadic observations done by factory operators. The thesis work is divided in three parts.

1.4.1 Testing and verification of automatic cold ingot detection

In the previous work, two approaches of detecting cold ingots in the body phase were proposed. From test results on several data sets, one of the approaches looks promising. A part of this thesis will aim to continue the previous work with the cold ingot detection. The continued work will be to implement the promising cold ingot detection algorithm in the live factory environment. This requires the algorithm to be modified to be practical in the live environment and its results need to be thoroughly examined to verify a high level of robustness and precision.

1.4.2 Automatic detection of structure loss in the body phase

Another problem is to develop an algorithm for detection of a different defect called structure loss in the body phase. This defect is a dislocation of the ingot and it renders the entire ingot unusable. It is a result of several factors such as temperature or entrapment of oxygen molecules at the edge of an ingot. This phenomenon should be detected as quickly as possible to minimize the time spent growing ingots with defects.

1.4.3 Automatic temperature detection in the neck stabilization phase

The third problem of this thesis is the detection of a desired state in an earlier stage of the Czochralski process, the neck phase. This phase is very temperature sensitive and the geometry of the silicon changes with cyclical temperature fluctuations in the melt. During such temperature cycles the ingot will enter several unwanted states unless the silicon melt is stabilized at the correct temperature. To ensure good initial conditions for crystal growth, the geometry of the solidified silicon should stay in the desired state for the duration of a full cyclical temperature fluctuation.

Automating detection of these phenomena will result in more precise and consistent error handling as well as reducing the workload of the factory operators. Faster detection of unwanted states allow in some cases the recovery of an otherwise unusable ingot, resulting in a net efficiency increase of the factory.

1.5 Limitations

The differing update frequency of the images between the live setup and the test setup proved to be a problem during the thesis work. Any time series data evaluation will differ between the test setup and the live setup, and some specific characteristics may occur in between the test data samples and thus be invisible on the test setup. To receive the data from the live setup I had to request specific data of specific phenomena, and some times I was not informed when the algorithm in the live setup had weird behaviour.

During the work with implementing an algorithm on the live setup in the factory a great challenge was that updating the software in the factory required coordination between me, the Sintef research team and personell on-site in the factory. Any variable that should be accesible for the real time logging system in the factory has to be created by *Prediktor AS* in their logging system, and connected through the main hardware of the puller setup.

An bug on the algorithm running in the factory was discovered and it took over one month before it was reported to the author. Communication has not always been smooth during the thesis work. Who should be contacted when different occurrences happen should have been planned in greater detail before the live factory testing began.

1.6 Thesis organization

The thesis is structured in a way such as to introduce all the required theory and tools used throughout the entire work, on all three separate problems. Some basic knowledge of techniques such as standard deviation and derivation is expected from the reader. Since the previous work project is very relevant for this thesis it will be introduced in a separate chapter as well. The three problems being handled in this thesis will have their own chapter which thoroughly explains the problem, the approach to solve the problem, the results of the approach and separate discussion sections. The end of this thesis will have a chapter dedicated to a conclusion of the three separate problems as well as recommendations for further work. All the code written for this thesis will be available in the appendix and referenced throughout the thesis so it can be reviewed if deemed necessary by the reader.

There will also be some images showing settings from *GUI* tools in *Scorpion Vision* to ensure the reader will be capable of recreating the setup used in this thesis.

Some sections are almost identical to those of the previous work as many of the same theoretical concepts and some of the same theory has to be introduced. The reused sections, with small variations, are **Sec. 1.0, Sec. 1.1, Sec. 1.3**, partly **Sec. 2.1.0**, **Sec. 2.1.1** and **Sec. 2.3.1**. The images within these sections are also taken from the previous work.

Chapter 2

Theory and tools

This chaper aims to inform the reader of the theory required to understand this thesis as well as a list of the tools used in the research.

2.1 Computer vision

A human looking around the world is able to perceive colour, light intensity and geometrical properties like depth, width and height almost instantaneously. We are able to understand what we are looking at through previous experiences with different scenary over many years. Programming a computer to understand imagery the same way is more difficult. With only one image the concept of depth will be gone from an image, and even if you use two images of the same scene the computer either needs some knowledge of the scene or perfect knowledge of the camera taking the images. Inherently computer vision is an inverse problem where we seek to recover some unknowns given insufficient information to fully specify the solution. Szeliski (2010)

Digitization of an image is done by dividing the image into a unit called a pixel or *px* for short. Digital cameras are always described with how many pixels they can have for each picture. A higher amount of pixels results in a better resolution. Depending on the image format every *px* is described by 8 or more bits. For a colour image with three channels, Red, Green and Blue (*RGB*) the image will usually be 24 or 32 bits. 8 bits for each colour channel and it can contain 8 bits for an alpha channel which describes the translucency of the image. This means that each of these channels can have a decimal value between $0 - 255$ describing the intensity of the respective channel. Since the maths in computer vision algorithms is heavy the image is often reduced to *grayscale* which is a reduction to 8 bits per pixel which solely represents the light intensity of a pixel in a range from black to white, or respectively $0 - 255$ light intensity. Since an image has both a width and a height the pixels are organized in a two dimensional array usually with a pixel corresponding to the same physical point of the image. This is usually denoted mathematically as:

$$I(x, y) \in [0, 255]. \tag{2.1}$$

Where the intensity I in position (x, y) contains a value between $0-255$ for a grayscale image. All images analyzed in this thesis will be in grayscale.

Even after converting to grayscale some computer vision algortihms can require so much processing power that they can only run within a Region of Interest (*ROI*). An *ROI* is just the region of the image which is of interest. This can be either a subset of the image or refer to the entire image. The *ROI* can either be dynamically determined by utilizing other computer vision techniques, or placed by design over an area. When the *ROI* is found, the algorithm can run within the boundaries of the region either by constraints or by simply cropping out that part of the image.

The success of computer vision is highly dependent on the scenario in which it is applied. Even though a lot of algorithms are highly general and work well in most scenarios, they require tuning of parameters to get a high probability of success. For some images and techniques, pre-processing of the image like grayscale conversion or contrast equalization is required to achieve a high success rate for most computer vision algorithms. Szeliski (2010)

2.1.1 Edge detection

Arguably the most iconic computer vision technique is edge detection. Within an image the interesting sections are often objects or geometrical properties linked to these objects such as position, length, width, depth or rotation. To be able to separate objects and decide their properties they have to be separated from the background. An edge detection can also be applied to a *ROI* if there's a specific area you want to find an edge within. There are many techniques for finding edges but this section will focus on how the Canny edge detector works due to its state-of-the-art status in popular open source software libraries such as *OpenCV*.

The first step of the Canny edge detector is to smooth the image within the *ROI* where the edge detector will be active. This is done to reduce noise, unwanted details and textures. A Gaussian filter of size $(2k+1)(2k+1)$ is convolved with the *ROI*. k is the variable which represents the size of the kernel. A larger filter will have a lower sensitivity to noise, but might result in the localization of the edges being less accurate. Canny (1986)

The filter values are given by:

$$\boldsymbol{H_{ij}} = \frac{1}{\pi\sigma^2} exp\left(-\frac{(i-(k+1))^2 + (j-(k+1))^2}{2\sigma^2}\right), 1 \leq i, j \leq (2k+1) \tag{2.2}$$

The filter is then applied by iterating the gaussian filter through the *ROI* and convolved with the image at every step. The next part of the process is the actual edge detection. For this there are several operators which can be used and Canny uses the Sobel Operator. This operator returns the value of the first derivate of the intensity function from **Eq. 2.1** in both horizontal and vertical direction.

$$G_x = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \left(\begin{bmatrix} 1 & 0 & -1 \end{bmatrix} * \boldsymbol{A}\right), \quad \text{and} \quad G_y = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \left(\begin{bmatrix} 1 & 2 & 1 \end{bmatrix} * \boldsymbol{A}\right). \tag{2.3}$$

where **A** is the intensity in the 3x3 pixel area around the desired pixel in the smoothed image.

Once the two directional gradients are found the true edge gradient value and direction can be determined by trigonometry:

$$G = \sqrt{G_x^2 + G_y^2} \quad \text{and} \quad \theta = atan2(G_y, G_x). \tag{2.4}$$

After the gradient is calculated the algorithm might find several directions from the evaluated pixel which give a positive edge response. This can cause the result to return unwanted edges. Non maximum suppression is then applied to reduce the result noise. This technique finds the local maxima gradient intensity while setting the other gradient values to zero. This ensures that only the strongest edge from the evaluated pixel is returned.

At this point edges in the picture are found and well represented but there might still be edges found due to noise or simply unwanted edges. Canny handles this problem by a step called hysteresis thresholding. It is a double threshold approach with an upper and a lower limit. Any pixel with gradient intensity above the upper limit is called a strong edge and is kept. Any pixel with a gradient intensity value between the upper and lower limit is called a weak edge and all its 8 neighbouring pixels are evaluated for a strong edge. If a strong edge is found within its neighbouring pixels the evaluated pixel will be relabeled as a strong edge and kept. If no strong edge is found in the closest neighbourhood of the evaluated pixel then the evaluated pixel will be discarded. All pixels with a gradient intensity value below the lower limit will be discarded as non-edges.

The upper and lower limit of the hysteresis threshold is application specific and is set by design based on a trial and error method to find the desired edge. Canny (1986)

Once all steps are completed the algorithm returns a binarized image with intensity 255 for the detected edges and 0 for the rest of the image. The result is then a black image with white lines describing the detected edges. The position and geometry of the edges can then be extracted and further analysed.

2.1.2 Machine learning and neural networks

As machine learning and neural networks are two enormous fields of study and not used very heavily in this thesis, this theory part will only scratch the utmost surface of the field.

Machine learning is a class of algorithms that allow a computer program to learn how to solve problems on its own.
A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E. Mitchell (1997)

Machine learning is based upon a statistical approach where patterns of data are classified through statistical dependence. Neural networks is a subset of machine learning which tries to mimic the human brain by modeling the data using graphs of *Artificial Neurons*.

These neurons are a mathematical model which tries to mimic how a neuron in the brain works.

The goal of neural networks is to find an unknown unlinear function f with known input x which results in a known output y. Goodfellow et al. (2016) The way this is done in practice is that a neural network model is created by stringing together layers of neurons with differing activation signals and sizes. The model is then trained upon a training set which is manually classified. For computer vision problems the classification could be images of tomatoes in one class, and images of apples in the other class. After the model has been trained, it can be validated on an image set of tomatoes and apples the model has not seen before and try to classify the images.

There are different kinds of neural networks, the Artificial Neural Network (*ANN*) is known as a *feed-forward neural network* which is a directed graph from input to output. Another type of neural network is the Recurrent Neural Network (*RNN*) which is a cyclical graph where some nodes output serve as input to themselves.

Typical problems that are solved by neural networks are classification problems or binary classification problems. Binary classification problems are a subset of classification problems where the goal is to decide if a specific phenomena either is, or is not present in the data. The regular classification is more complex and tries to classify several types of phenomena from the data. Machine learning with neural networks is the current state of the art technology in a wide range of industries such as ad targeting, finance trends, computer vision, search engine results and speech recognition, among many other fields.

Overfitting and underfitting

When training a neural network model and comparing their test and validations scores it is important to note how well the model is doing on both. There are two concepts, called overfitting and underfitting. Overfitting happens when the neural network finds too many complex relationships in the data such as sampling noise which can exist in the training data but not necessarily in the validation data. This is spotted by the model scores on the validation data reaching a peak value and then starts to drop down. It means that the connections the model is able to find between the data and the result is not the connection it is supposed to find. Srivastava et al. (2014)

Underfitting on the other hand is a situation where the validation scores are higher than the scores on the training data which means that the model is not able to correctly model the true patterns of the images correctly as a result of either an incomplete data set or a weak neural network model for the specified problem.

The problems of overfitting and underfitting are illustrated in **Fig. 2.1**.

2.2 Frequency analysis

Frequency analysis aims to break down a complex signal into its components at various frequencies. Tao et al. (2007) Through various techniques such as Fourier transforms or wavelet transform a continuous-time function is transformed to the frequency domain to look at what happens at specific frequencies instead of looking at when something happens in the time domain.

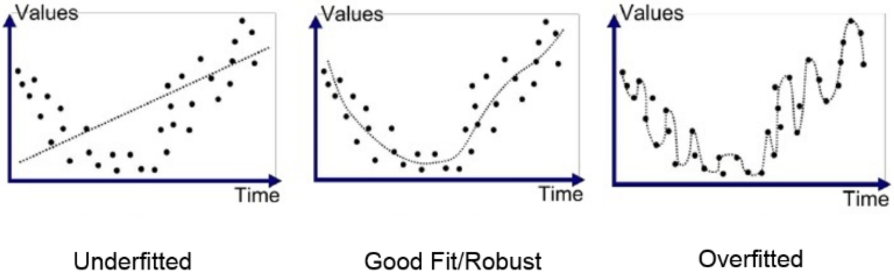


Figure 2.1: A simplistic illustration of over- and underfitting in machine learning. Liew (2016)

2.2.1 Fourier transform

The Fourier transform decomposes a time function into the frequency components. Using the Fourier transform can allow a look into the frequency components responsible for the periodic changes in a time series signal. The Fourier transform uses the fact that any arbitrary set of data can be represented by a possibly infinite series of sines and cosines. A disadvantage of a Fourier expansion is that it only has frequency resolution but no time resolution. Valens (1999)

2.2.2 Continuous wavelet transform

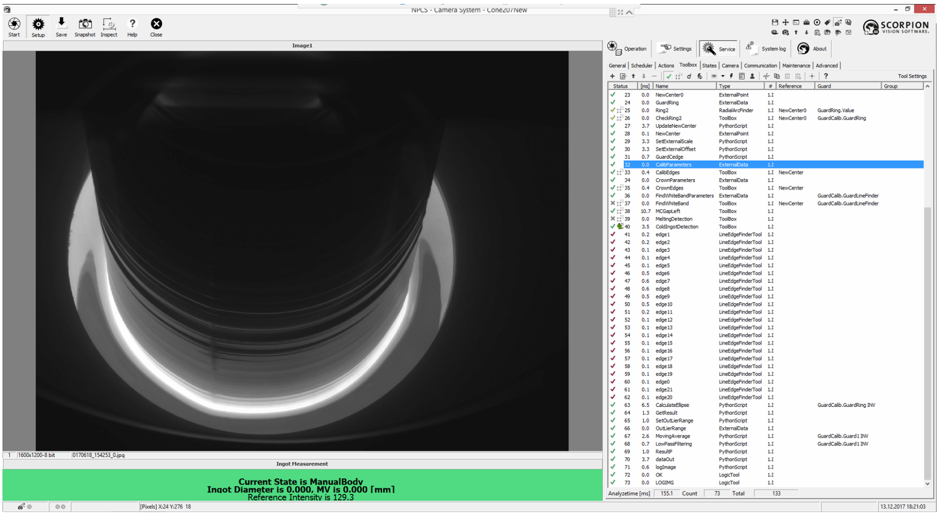
A wavelet is a wave that oscillates with an amplitude that begins at zero, increases and then decreases back to zero again. Wavelets are the solution to the Fourier transforms shortcoming in that it allows a relationship between frequency resolution and their respective time resolution. The wavelet transform results in a collection of time-frequency representations of the signal with different resolutions, or levels of detail. Valens (1999)

2.3 Tools and software libraries

Many computer vision algorithms and mathematical analysis tools are already implemented in various libraries and frameworks. Using these tools reduces the time to be efficient if chosen and used properly when necessary. This section will introduce the frameworks and software libraries used throughout this thesis.

2.3.1 Scorpion vision framework

Most of work in this thesis will be conducted in an environment which already has a computer vision setup with a camera and a PC with software to analyze the images. This software is a third-party machine vision solution developed by Tordivel and the software is called Scorpion. Scorpion is a computer vision software framework with a graphical user interface *GUI*. The framework has functionality such as an integrated picture slideshow functionality, script implementation without needing to compile the code, and ready-to-use



Status	[ms]	Name	Type	#	Reference	Guard	Group
✓ 23	0.0	NewCenter0	ExternalPoint	1.I			
✓ 24	0.0	GuardRing	ExternalData	1.I			
✓ 25	0.0	Ring2	RadialArcFinder	1.I	NewCenter0	GuardRing.Value	
✓ 26	0.0	CheckRing2	ToolBox	1.I	NewCenter0	GuardCalib.GuardRing	
✓ 27	3.7	UpdateNewCenter	PythonScript	1.I			
✓ 28	0.1	NewCenter	ExternalPoint	1.I			
✓ 29	3.3	SetExternalScale	PythonScript	1.I			
✓ 30	3.3	SetExternalOffset	PythonScript	1.I			
✓ 31	0.7	GuardCedge	PythonScript	1.I			
✓ 32	0.0	CalibParameters	ExternalData	1.I			
✓ 33	0.4	CalibEdges	ToolBox	1.I	NewCenter		
✓ 34	0.0	CrownParameters	ExternalData	1.I			
✓ 35	0.4	CrownEdges	ToolBox	1.I	NewCenter		
✓ 36	0.0	FindWhiteBandParameters	ExternalData	1.I		GuardCalib.GuardLineFinder	
✗ 37	0.0	FindWhiteBand	ToolBox	1.I	NewCenter	GuardCalib.GuardLineFinder	
✓ 38	10.7	MCGapLeft	ToolBox	1.I			
✗ 39	0.0	MeltingDetection	ToolBox	1.I			
✓ 40	3.5	ColdIngotDetection	ToolBox	1.I			
✓ 41	0.2	edge1	LineEdgeFinderTool	1.I			
✓ 42	0.2	edge2	LineEdgeFinderTool	1.I			
✓ 43	0.1	edge3	LineEdgeFinderTool	1.I			
✓ 44	0.1	edge4	LineEdgeFinderTool	1.I			
✓ 45	0.1	edge5	LineEdgeFinderTool	1.I			
✓ 46	0.5	edge6	LineEdgeFinderTool	1.I			
✓ 47	0.6	edge7	LineEdgeFinderTool	1.I			
✓ 48	0.6	edge8	LineEdgeFinderTool	1.I			
✓ 49	0.5	edge9	LineEdgeFinderTool	1.I			
✓ 50	0.5	edge10	LineEdgeFinderTool	1.I			
✓ 51	0.2	edge11	LineEdgeFinderTool	1.I			
✓ 52	0.1	edge12	LineEdgeFinderTool	1.I			
✓ 53	0.1	edge13	LineEdgeFinderTool	1.I			
✓ 54	0.1	edge14	LineEdgeFinderTool	1.I			
✓ 55	0.1	edge15	LineEdgeFinderTool	1.I			
✓ 56	0.1	edge16	LineEdgeFinderTool	1.I			
✓ 57	0.1	edge17	LineEdgeFinderTool	1.I			
✓ 58	0.1	edge18	LineEdgeFinderTool	1.I			
✓ 59	0.1	edge19	LineEdgeFinderTool	1.I			
✓ 60	0.1	edge0	LineEdgeFinderTool	1.I			
✓ 61	0.1	edge21	LineEdgeFinderTool	1.I			
✓ 62	0.1	edge20	LineEdgeFinderTool	1.I			
✓ 63	6.5	CalculateEllipse	PythonScript	1.I		GuardCalib.GuardRing INV	
✓ 64	1.3	GetResult	PythonScript	1.I			
✓ 65	1.0	SetOutLierRange	PythonScript	1.I			
✓ 66	0.0	OutLierRange	ExternalData	1.I			
✓ 67	2.6	MovingAverage	PythonScript	1.I		GuardCalib.Guard1 INV	
✓ 68	0.7	LowPassFiltering	PythonScript	1.I		GuardCalib.Guard1 INV	
✓ 69	1.0	ResultP	PythonScript	1.I			
✓ 70	3.7	dataOut	PythonScript	1.I			
✓ 71	0.6	logImage	PythonScript	1.I			
✓ 72	0.0	OK	LogicTool	1.I			
✓ 73	0.0	LOGIMG	LogicTool	1.I			

Analyzetime [ms] 155.1 Count 73 Total 133

1 1600x1200-8 bit 0170618_154253_0.jpg

Current State is ManualBody
Ingot Diameter is 0.000, MV is 0.000 [mm]
Reference Intensity is 129.3

[Pixels] X:24 Y:276 18

13.12.2017 18:21:03

Figure 2.2: Full view of Scorpion with image slides to the left and toolbox overview on the right.

implementations of a wide variety of computer vision algorithms. This allows updating scripts and variable values in real time which can be a very powerful tool while designing algorithms. In the current version of Scorpion the programming language integrated with the framework is Python 2.7.

Since this setup is already in use in the factory there are many toolboxes and scripts implemented and ready to use for different types and phases of the Czochralski process. The following subsections will introduce the most important tools and toolboxes used in this thesis.

2.3.2 LineFinder2

The LineFinder2 tool is an edge detector which uses first or second order derivatives to detect light intensity variations, much in the same way the *Canny* edge detector does, along tracelines within an *ROI*. The tool has a lot of threshold settings and it returns the position, angle and several more properties of the detected edge.

2.3.3 FindWhiteBand toolbox

The FindWhiteBand toolbox is a set of tools and scripts that place small edge detectors on the meniscus all the way around the ingot as seen in **Fig. 2.3**. The position of these edges are used for calculating the diameter of the ingot and it is used in the *ADC* feedback loop.

The entire toolbox is shown in **Fig. 2.4** and it contains several modules. The guards at the beginning are logic controllers to ensure the toolbox only runs in the correct phases of the process. The FindWhiteBandParameters is a tool containing the initial condition data for the toolbox. *IncomingRef*, *CenterRef*, *SearchData* and *SearchRef* are positional tools ensuring that the FindWhiteBand toolbox is centered relative to the center of the

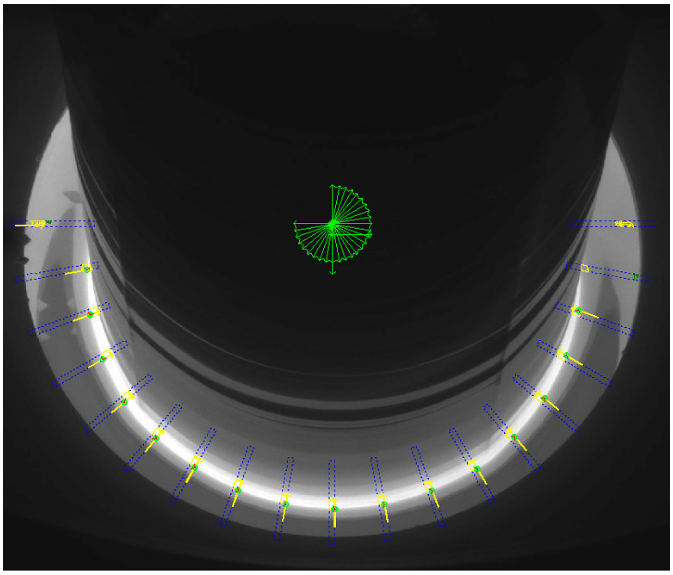
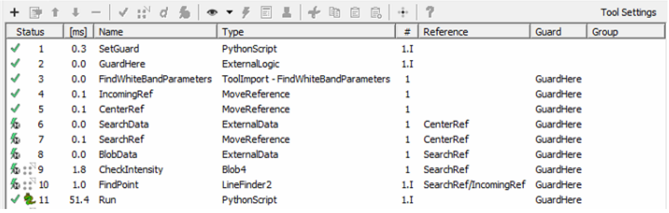


Figure 2.3: Edge detectors placed by the FindWhiteBand toolbox.



Status	[ms]	Name	Type	#	Reference	Guard	Group
✓ 1	0.3	SetGuard	PythonScript	1.I			
✓ 2	0.0	GuardHere	ExternalLogic	1.I			
✓ 3	0.0	FindWhiteBandParameters	ToolImport - FindWhiteBandParameters	1		GuardHere	
✓ 4	0.1	IncomingRef	MoveReference	1		GuardHere	
✓ 5	0.1	CenterRef	MoveReference	1		GuardHere	
6	0.0	SearchData	ExternalData	1	CenterRef	GuardHere	
7	0.1	SearchRef	MoveReference	1	CenterRef	GuardHere	
8	0.0	BlobData	ExternalData	1	SearchRef	GuardHere	
9	1.8	CheckIntensity	Blob4	1	SearchRef	GuardHere	
10	1.0	FindPoint	LineFinder2	1.I	SearchRef/IncomingRef	GuardHere	
✓ 11	51.4	Run	PythonScript	1.I		GuardHere	

Figure 2.4: The entire FindWhiteBand toolbox.

ingot and updates its coordinate reference system accordingly. The *BlobData* and *CheckIntensity* tools are tools to set up a *blob* which is an *ROI* with constant or approximately constant properties, such as light intensity. The blob is set up to find the highest threshold area within the search area shown by a blue dotted line in **Fig. 2.3**. It then returns the threshold used to isolate the highest intensity area. This threshold is dynamically applied to the FindPoint tool which is a *LineFinder2* algorithm. This ensures that even if the light intensity conditions of the meniscus changes, the *LineFinder2* algorithm will always find the meniscus edge if it is possible. The script called *Run* handles all the internal logical of the toolbox, combining all the tools and dynamically updating their values to place the edge detectors at their correct positions and return the position of the found edges.

2.3.4 SciPy

SciPy is a collection of open-source software for mathematics, science and engineering. This thesis uses functionality from three of the open-source libraries contained in the SciPy package. NumPy, MatplotLib and the SciPy library. Jones et al. (2001–)

SciPy is a fundemental library for scientific computing.

2.3.5 Matplotlib

Matplotlib is a 2D plotting library for Python that simulates the functionality of plotting in MATLAB for Python projects. It produces publication quality figures and interactive environments across platforms. Hunter (2007)

2.3.6 Numpy

NumPy is a fundamental package for scientific computing with Python. It has functionality to handle N dimensional array objects, basic linear algebra and basic fourier transforms. It is also developed with a C/C++/Fortran base which makes data handling several magnitudes faster than regular Python which makes it very useful for optimization. Canny (1986)

2.3.7 TensorFlow

TensorFlow which is a machine learning library widely used for speech recognition, computer vision, robotics, information retrieval, natural language processing, geographic information extraction and computational drug discovery. It was developed at Google and released as open-source. Abadi et al. (2015)

2.3.8 Keras

Keras is neural network API which is an abstraction of lower level machine learning libraries. For this thesis Keras is running on top of TensorFlow. Keras' API is an abstraction which focuses on user friendlisness for the developer which improves the speed of which an application can be developed and debugged. Chollet et al. (2015)

Chapter 3

Previous work: Cold Ingot Detection

The algorithm which was tested to be the most promising one for detecting a cold ingot in the body phase is called the meniscus angle method and it uses geometrical changes in the meniscus between the silicon melt and the ingot as an indicator of coldness. The method is based upon the following hypothesis which was based upon the geometrical changes visible in Figure 3.1. The code for this algorithm is shown in the *Appendix*.

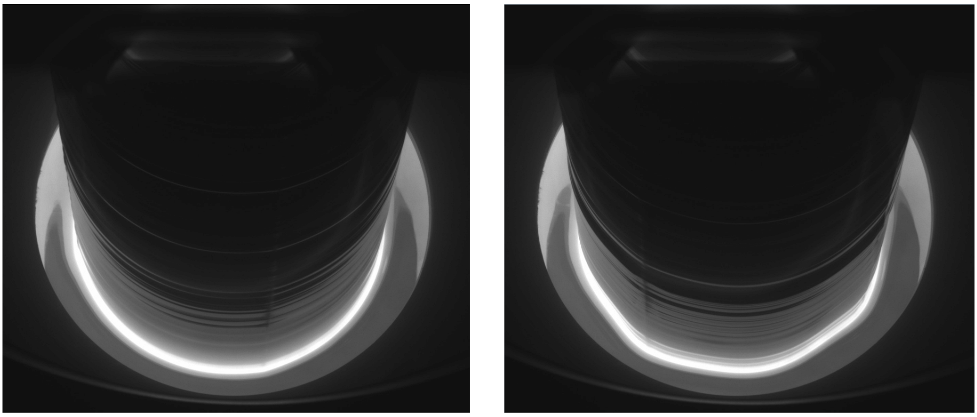
Hypothesis:

- The angle of a line segment along the meniscus will vary more often and more intensely when the ingot is cold compared to when it is in the desired state.
- The intensity of angle fluctuations in line segments along the meniscus sampled over several frames should increase enough to be measured when the ingot becomes cold.

Implementation of the algorithm was done by adding three edge detectors as shown in Figure 3.2. The detectors all return a line segment of the detected edge and their respective angle. The angle for all three edge detectors are stored for $N = 30$ frames. For each detector the standard deviation of the angle over N frames is calculated and then averaged over all three gathered data sets for redundancy. After the array of 30 frames is initially filled up the rest of the updates to the array is done as a first in first out (*FIFO*) queue. This results in the algorithm continuously updating its standard deviation for every new frame which is evaluated after the initial 30 frames.

The result from the algorithm is a floating point value which describes how fast the geometrical form of the meniscus changes. A higher value represents a colder ingot while a lower value represents an ingot in the desired state. This was run over three independent data sets and all three tests succeeded.

The algorithm was developed and implemented on a test setup on the Scorpion software introduced in **Sec. 2.3.1** which allows the user to run computer vision algorithms



(a) View of the ingot in the desired state.

(b) View of the ingot when it becomes cold.

Figure 3.1: Side-by-side comparison of an ingot in the desired state and a cold ingot.

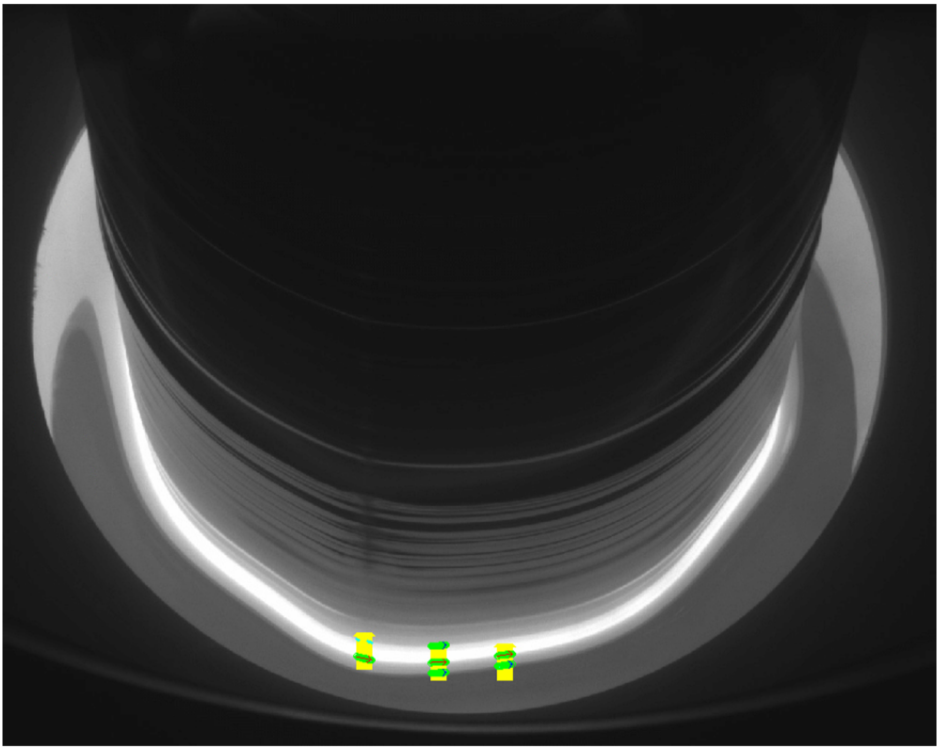


Figure 3.2: A view of the three edge detectors placed on the meniscus.

on a slideshow of images from the process. The images used for designing and testing of the algorithm are taken every 20 seconds. In the live setup a new photo is taken as fast as the Scorpion software can process it, currently five times per second. This means that information is lost between testing on the live setup and the simulation setup.

Chapter 4

Test of Cold Ingot Detection in a live environment

In the fall project a proof of concept algorithm was designed and tested in a test environment. This chapter will focus on ensuring proper functionality and robustness of the detection algorithm. To accomplish this behaviour the algorithm has to be tested in the live environment in the factory.

This chapter describes the changes made to interface the algorithm with the live setup in the factory, the implementation of the algorithm, tuning the algorithm and provides a discussion of the results obtained and possibilities for improvement.

4.1 System implementation

In preparation of implementing the algorithm in the live setup in the factory, a general understanding of the system implementation should be acquired.

The overview of the data flow in the system is shown in Figure 4.1. The components in the figure are described below.

The camera is the actual camera mounted on the furnace. It is a Sony XCG-U100E with a pixel resolution of 1600 x 1200px and is specifically designed for machine vision applications. Sony (2016)

The Camera PC is the computer where the images are run through the Scorpion Vision software. This is where all the computer vision algorithms run and measurements are extracted from the image. Most of this thesis work is done within this component, but some care has to be taken to be able to communicate with the rest of the system.

WinCC is a scalable process-visualization system for visualizing, monitoring and operating processes in a plant. The software is widely used in the industry and it is created by Siemens.Siemens (2016) For this setup it is the main link ensuring smooth communication between all the components.

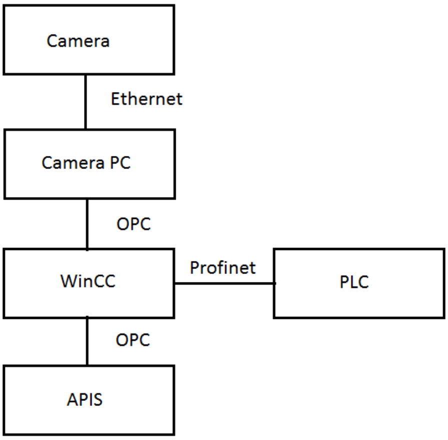


Figure 4.1: System data flow chart.

The Programmable Logic Controller *PLC* performs the conversion from a digital control signal to an analog signal suitable for sending to an actuator. This is the component which can actuate the furnace or the connected alarm system from measurements done by the Camera PC component.

APIS is a real-time software platform for industrial applications. It streamlines information sharing, logging and visualization from many components in the plant. It is a software designed by Prediktor and it is used in over 600 plants throughout Norway.Prediktor (2018) APIS is the main tool used by the Sintef researchers involved in this project and the process engineers at the plant to monitor the pullers. It keeps track of all digital variables and measurements from the furnace which can be dynamically put into the same graph for analyzis as shown in **Fig. 4.2**.

The ethernet connection is using a standard TCP/IP protocol which is connected by cable, while OPC is an industry standard protocol for communication between components. Profinet is a communications protocol developed by *Profibus & Profinet International* which specializes in delivering data under tight time contraints. The Profinet protocol is used as a link between *WinCC* and the *PLC*. Profinet runs on top of ethernet and allows open and fast communication based on TCP/IP in parallel connections over the same cable.

The full data flow of an image then becomes clear. A photo is taken by the camera module. It is sent through ethernet to the camera PC which runs the image through *Scorpion* to analyze the picture. Scorpion keeps its own local logic about which states to run depending on what it sees from the camera frames and thus the output depends on its internal logic. The output from *Scorpion* are return values from algorithms which are written through the OPC link to *WinCC*. WinCC then sends this value to the equivalent variable in *APIS* for logging and easy visualization and also to the *PLC* over Profinet if there should be a physical response, for example a light going on. So for a value found by a computer vision algorithm to become available for the rest of the system, it has to be written to an *OPC* variable which is then further treated by WinCC. This allows the PLC to use and

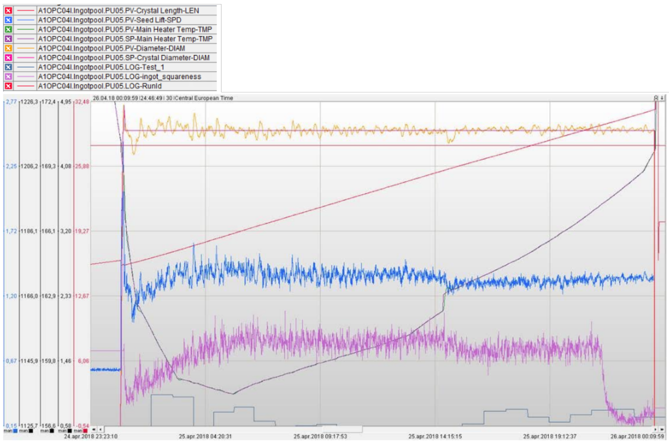


Figure 4.2: An example view of the data visualization capabilities of APIS.

APIS to log the value.

4.2 Preparations for implementation

To implement the algorithm some changes from the Project version from 2017 is made. Both the old and the new improved code can be viewed in the *Appendix*. In the previous project the standard deviation was calculated in Python's standard library. To optimize the calculation, Numpy's functionality for calculating standard deviation was used as its operations are several magnitudes faster than the Python standard library. The original central meniscus edge detector was placed statically over the meniscus, which is not ideal since the exact placement may vary between ingots. The positioning is changed to place the edge detectors according to a position dynamically found by the *FindWhiteBand* tool in Scorpion to avoid changing position on the meniscus between ingots. The two other edge detectors now use the center detector position as reference and place themselves relative to it. Since Scorpion allows code to run without needing to compile, a functionality that would handle a size decrease in the arrays used to calculate the standard deviation of the angle along a line segment. This functionality is implemented to allow a decrease in array size in the middle of the run without introducing any errors. This functionality ensures that the already saved line angle array is purged until an array of the new set size is filled. To interface with the other systems a function to write the resulting value over the *OPC* protocol also had to be enabled. The *OPC write* function is a heavy function which requires $7 - 10ms$ to run each time it is called. A counter was implemented to ensure the *OPC write* function would only be called every *X* frames. This ensures that the extra $7 - 10ms$ overhead of the write happens on a slower interval than images being evaluated. This will result in the result being written to the *OPC* link will be delayed, but the interval can be tuned as needed by changing the amount of frame, *X*, to iterate over between every write. The cold ingot detection algorithm writes to an *OPC* variable named *ingot_squareness* that takes *integer* type values. Since the algorithmic results of standard deviation are floating

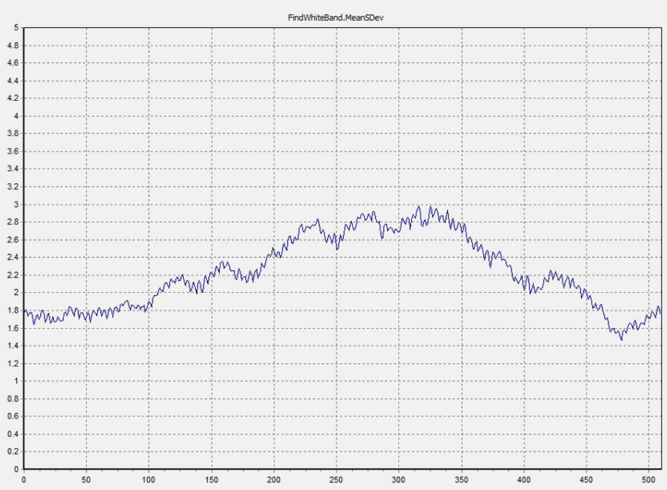


Figure 4.3: Standard deviation over three hours of sample data on an ingot forced cold by manual temperature adjustment.

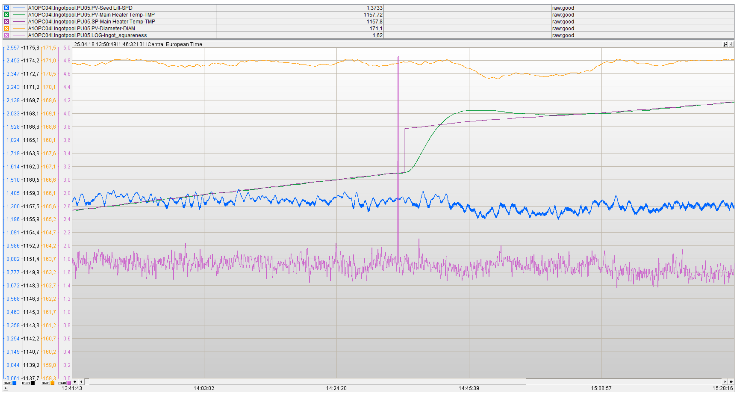
points, a conversion is performed by multiplying the value by 1000 to preserve 3 decimal places precision.

4.3 First test period

For the first month the only test of the algorithm was a forced test performed on March the 2nd. An operator lowered the temperature manually to make the ingot cold. The standard deviation of the meniscus angle clearly became larger as the ingot became colder as seen in **Fig. 4.3**.

This ingot was forced cold manually until the *ingot_squareness* hit 3.0 which is the alarm threshold. The corresponding alarm activated and the temperature was adjusted back up again manually which let the ingot grow back into a cylindrical shape. The graph in **Fig. 4.3** never actually hits 3.0. This is because the graph is sampled from a simulation of the data on the test setup. The data set only contains samples in 20 second intervals instead of 5 per second which is the update frequency in the live setup, which made the *ingot_squareness* value reach 3.2 and thus triggered the alarm.

As the algorithm kept running throughout March and April there were reported some problems with sudden spikes in the *ingot_squareness* variable at random points during the body phase, and whenever the ingot was close to finishing the body phase. The operators in the factory would try to adjust the temperature but after several occurences they started discarding the alarm. The threshold value for *ingot_squareness* to activate an alarm was raised from 3.0 to 4.0 by a Sintef researcher on the request of process engineers from the factory without consulting the author. This was intended to avoid operators manually discarding the alarm every time it reported a false positive. An example of a spike is seen in **Fig. 4.4**. A temperature adjustment was performed by an operator as a response to the



☒	A1OPC04I.Ingotpool.PU05.PV-Seed Lift-SPD	1,3733	raw:good
☒	A1OPC04I.Ingotpool.PU05.PV-Main Heater Temp-TMP	1157,72	raw:good
☒	A1OPC04I.Ingotpool.PU05.SP-Main Heater Temp-TMP	1157,8	raw:good
☒	A1OPC04I.Ingotpool.PU05.PV-Diameter-DIAM	171,1	raw:good
☒	A1OPC04I.Ingotpool.PU05.LOG-ingot_squareness	1,62	raw:good

Figure 4.4: Spike in *ingot_squareness* in the middle of the body phase. The step response in *SPMain HeaterTMP* is a manual temperature adjustment.

alarm activation due to the spike, seen in the step response in *SPMain HeaterTMP*. This adjustment is detrimental for the process as it throws off the reference temperature for the control system and may introduce undesirable situations later in the process.

The 12th of April an ingot became cold without being forced manually. At this point the alarm threshold for *ingot_squareness* was still set to 4.0 and it was at this point the author was made aware that the random value spikes was a problem in the live setup. Since the threshold was set to 4.0 the ingot became very cold. The ingot lost its structure due to the thermal stress and had to be discarded. The period when the ingot became cold is simulated with data from the factory on the test setup with 20 second intervals between every image and is shown in **Fig. 4.5**.

This incident resulted in a temporary fix where the alarm threshold for *ingot_squareness* was reduced to 3.0 again and a timer was implemented which only allowed one alarm every 20 minutes to activate while the root of the thresholding problem would be worked on.

Analyzing the image data set around the time of the spikes shows that there are two main problems. When the meniscus edge detectors fails to find an edge they write 0.0 to their respective array instead of returning nothing. This causes a huge spike in standard deviation since the line segment angle usually fluctuates around 90° and every standard deviation evaluation is done on a horizon of 30 frames. This means that a 90° change compared the the regular $1 - 3°$ makes a big difference. The other error was that at the end of the body phase the meniscus shrinks and the edge detectors finds the edge above the meniscus as seen in **Fig. 4.6b** or the next edge below the meniscus as seen in **Fig. 4.6a**. Finding a different edge results in the line segment angle having a slightly different angle than the meniscus angle thus introducing inconsistencies in the standard deviation measurement.

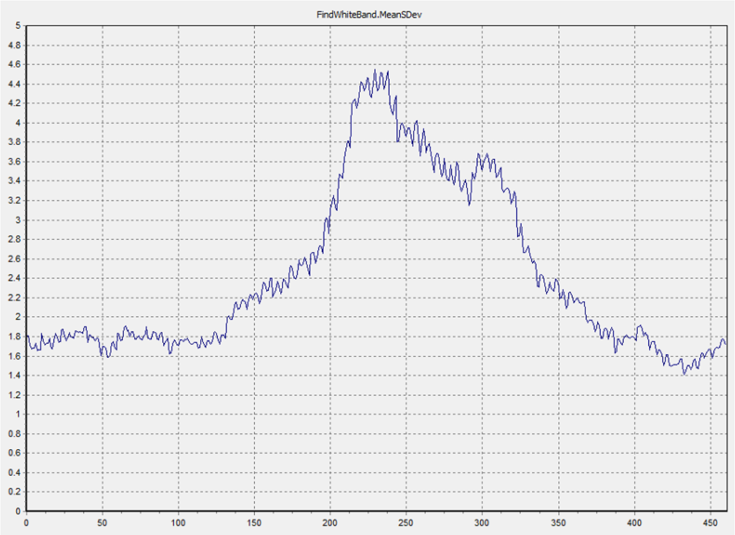
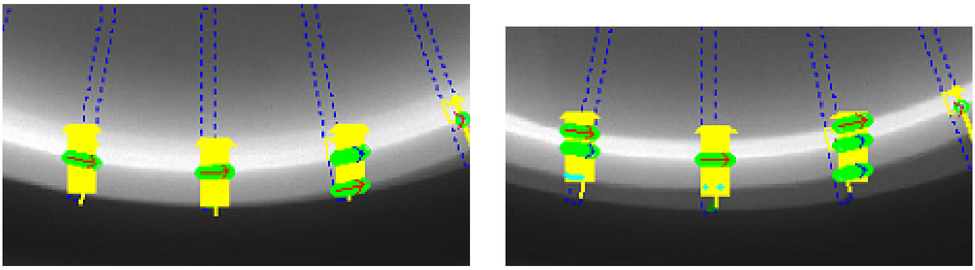


Figure 4.5: *ingot_squareness* for the very cold ingot with alarm threshold set to 4.0 on the 12th of April 2018.



(a) The right meniscus edge detector finding the wrong edge below the meniscus indicated by red arrow.

(b) The right and left meniscus edge detector finding the wrong edge above the meniscus indicated by red arrows.

Figure 4.6: Examples of the meniscus edge detector finding wrong edges at the end of the body phase.

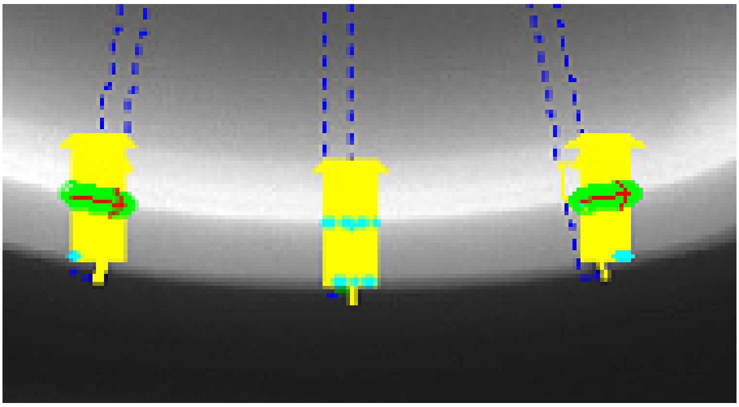


Figure 4.7: The center meniscus edge detector not able to find an edge indicated by no red arrow.

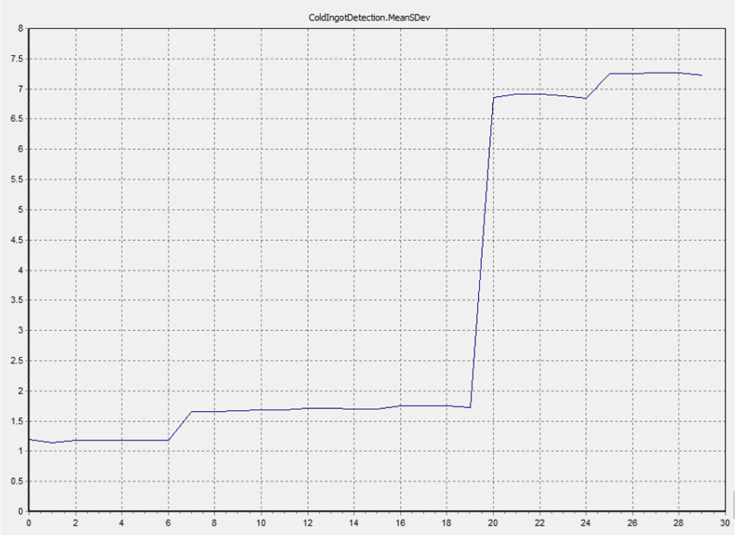
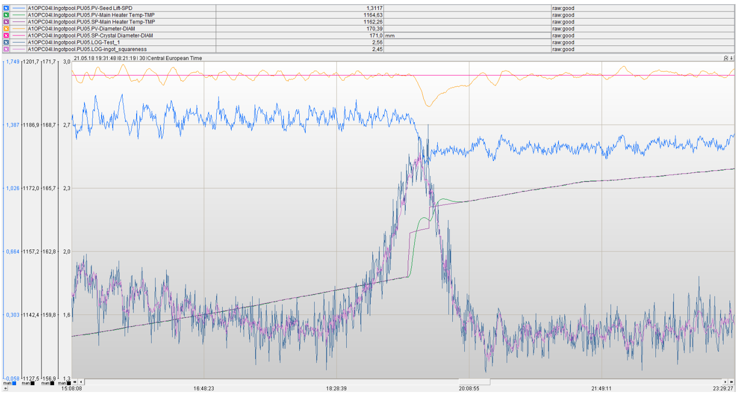


Figure 4.8: The standard deviation spiking over a 30 image dataset when edges are not found.



☒	A1OPC04I.Ingotpool.PU05.PV-Seed Lift-SPD	1,3117		raw:good
☒	A1OPC04I.Ingotpool.PU05.PV-Main Heater Temp-TMP	1164,63		raw:good
☒	A1OPC04I.Ingotpool.PU05.SP-Main Heater Temp-TMP	1162,26		raw:good
☒	A1OPC04I.Ingotpool.PU05.PV-Diameter-DIAM	170,39		raw:good
☒	A1OPC04I.Ingotpool.PU05.SP-Crystal Diameter-DIAM	171,0	mm	raw:good
☒	A1OPC04I.Ingotpool.PU05.LOG-Test_1	2,56		raw:good
☒	A1OPC04I.Ingotpool.PU05.LOG-ingot_squareness	2,45		raw:good

Figure 4.9: *APIS* graph which shows the time period where *ingot_squareness* indicates the ingot becoming cold.

4.4 Tuning

After identifying the problems with the current algorithm, some changes were made. The script that calculates the standard deviation has added checks to see whetever a meniscus edge detector has a succesful status. Initially while filling the data set the script only allows measurements when all three edge detectors find the edge. The script then checks the status of the currently measured edge before replacing a value in the standard deviation array. This results in a new value being appended only for the edges which have a succesful edge detected.

The solution to finding wrong edges in the end of the body phased was solved by using an intensity check found in the FindWhiteBand toolbox which looks at the same region. It checks the intensity within a blob and uses the intensity to dynamically update the limits the edge detector should operate within.

4.5 Late detection of cold ingots

The 24th of May a report from the a process engineer at the factory stated that several times operators had seen the ingot becoming cold without the alarm going off in time. It was often characterized by the *ingot_squareness* value decreasing before increasing again as seen in **Fig. 4.9**. The image set from the relevant time period was sent for analysis.

Visually the ingot seen in **Fig. 4.9** is as cold as other ingots which have resulted in the *ingot_squareness* increasing above the threshold at 3.0. When the *ingot_squareness* decreases before the spike the ingot looks very elliptical without any signs of having a jagged geometry which indicates a cold ingot. This means that the decrease in value before the spike is most likely just a coincidence and can not necessarily be used to help identify cold ingots. There seems to be two plausible explanations. Either the meniscus line detectors seen in **Fig. 3.2** are placed in such a way that the geometry of the cold ingot cancel out the measurements. For example the left edge always reporting a low change of standard deviation in its angle at the same time as the right edge is reporting a big change, thus cancelling eachother out. This seems unlikely as over the 30 image frames used to calculate the standard deviation the cold ingot geometry should affect all three edge detectors. This is very difficult to analyze on the test setup due to the 20 seconds between images.

Another possible explanation is that the standard deviation increases more slowly if the ingot becomes cold at a slower rate. If the ingot becomes colder at a slower rate the outliers from the mean in the standard deviation may decrease as the mean increases. If the changes are happening slowly enough the mean angle of the data set can catch up with the outliers thus the reported standard deviation might be lower than expected.

The problem might also stem for a specific sampling phenomenon depending on a correlation between the rotations per minute of the ingot in the puller and the sample size and frequency of the algorithm. This has to be analyzed in the factory environment with real time data.

4.6 Discussion

There has been some problems with the cold ingot detection after the implementation. Firstly the bug which made the *ingot_squareness* value spike at random intervals through the process. This bug has been fixed on the test setup but due to the author receiving information about the problem over a month after the implementation and with co-supervisor John Atle on leave it has been difficult to find a time to implement the new algorithm. By the 4th of June when this is written the algorithm is yet to be implemented and receiving any data from the implementation in time of the thesis deadline will be impossible.

After the initial bug reports there seems to have been discovered a problem where the ingot becomes cold without the *ingot_squareness* exceeding the threshold at 3.0. This seems to be a result of an ingot becoming cold more slowly than tests done in the previous work when the algorithm was developed. This problem needs to be analyzed on the live setup with a full data set of images without the 20 second interval to properly decide if the edge detectors should change their placement or if the threshold for the cold ingot alarm should be lowered. Through correspondence with process engineer Jeroen Van Delft it has been decided that for the next iteration of the algorithm some more functionality should be added. Specifically this functionality should facilitate an *OPC write* for the individual edge detectors to analyze their behaviour when the phenomenon of undetected cold ingots occur.

Chapter 5

Detection of structure loss in body phase

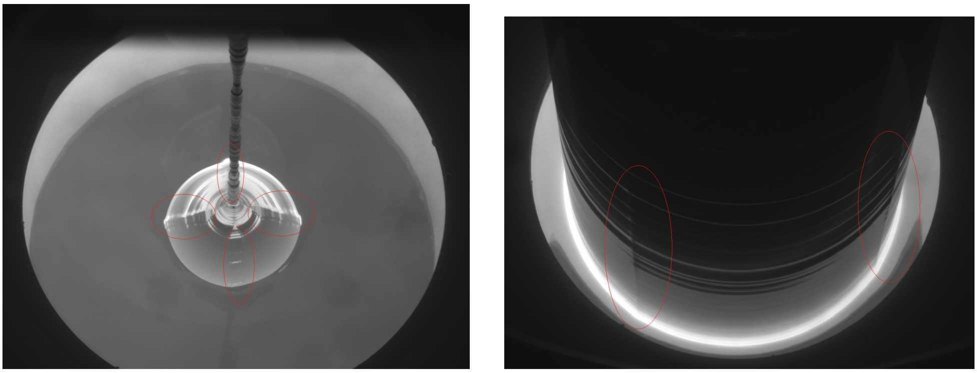
Structure loss is an issue that limits the potential yield of crystals grown by the Czochralski process in the photovoltaic industry. This issue occurs by generation and propagation of crystal dislocations during the pulling process. Lanterne et al. (2016) While the Cz crystal is growing, four nodes will form symmetrically at the edge of the crystal as seen in **Fig. 5.1**. These nodes contain the homogenous form of the crystal which allows the crystal to be grown to a specified size. When the crystal becomes dislocated it can be seen by the loss of one or more of the four nodes which are growing along the edge of the crystal. This phenomenon is knows a structure loss and is often a result of thermal stress due to cooling of the ingot, or the crystallization of silicon around a particle which grows as a bump on the edge of the ingot. Lanterne et al. (2016)

State of the art detection of structure loss is visual inspection by operators looking into the furnace on their rounds every 30 minutes. The visual cues to decide if an ingot is experiencing structural loss is by looking for the four nodes on the rotating ingot. The nodes are shown in **Fig. 5.1**. When an ingot is experiencing structural loss, the affected node will stop growing along the ingot and the bump which is usually seen in the meniscus will disappear.

As **Fig. 5.2** illustrates, the ingot will continue growing without the nodes present. This means that for an early detection of structure loss the observer should look as closely as possible to the liquid silicon.

5.1 Problem specification

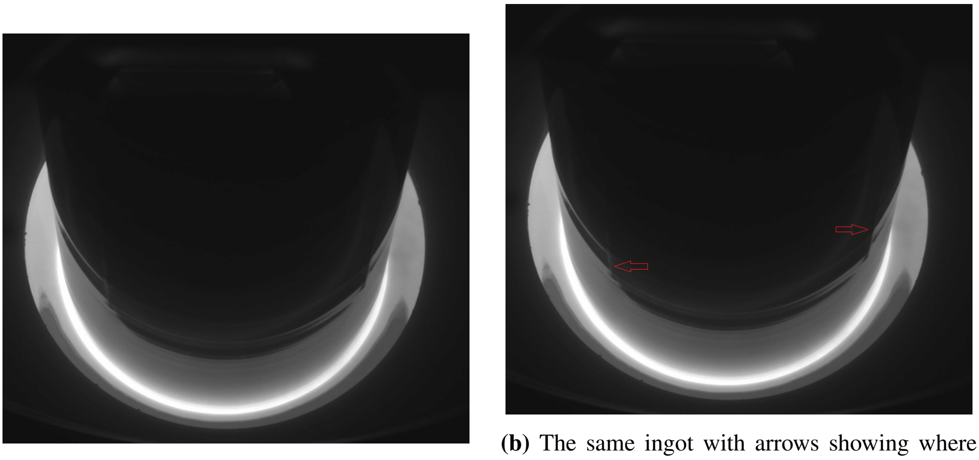
On the surface the problem is simple. If an operator sees that a node or more is missing, then a computer should be able to do the same as fast, or faster. The ingot has four nodes symmetrically positioned to eachother on the edges of the cylindrical ingot. As the ingot rotates at most two of the nodes will be visible at any given time. When the node is in



(a) View of the four nodes marked in red in the crown phase.

(b) View of two nodes marked in red in the body phase. The ingot hides the other two nodes.

Figure 5.1: The nodes on a growing ingot.



(a) A dislocated ingot.

(b) The same ingot with arrows showing where the nodes stopped growing.

Figure 5.2: An ingot experiencing structural loss.

the center front of the ingot then only one node wil be clearly visible. If a node can be automatically detected then the number of frames, or time between each detected node can be used to identify structure loss in the ingot. So the main focus of this approach will be to detect the nodes.

For a computer vision based detection approach either a light intensity change or geometrical identity when the node passes, or should pass if it's non existing, in the camera view has to be discovered.

In the images from **Fig: 5.3** it is clear for the human eye that the nodes are intact because the nodes are clearly visible along the edge of the ingot all the way down to the meniscus. This is due to the clear break of light from one side of a node to another side of the node as illustrated in **Fig. 5.4**. But there are some problems with this way of detecting the existence of a node through the light breaking property. Firstly the break of light is a lot clearer some distance above the meniscus as illustrated in **Fig. 5.2**. Using this property to detect structure loss would result in a slow detection. Another factor is that the lines of shadows forming around the node are not consistent throughout the body phase of one ingot, nor is it consequent between separate ingots. To reduce the time for detection the algorithm should look closer to the meniscus, but as shown in **Fig. 5.4** the difference in light intensity around the node a small distance above the meniscus is very small.

Another approach is to use the change in geometry when a node passes a point during the rotational movement. **Fig. 5.5b** shows that when a node passes in the center of the image, a dip slightly down into the meniscus can be observed. This also makes the bottom edge of the meniscus bulge as seen in **Fig. 5.5a**. So it might be possible to deduce if a node exists by looking at the geometrical changes in the meniscus.

There is already implemented measurements for the geometrical changes in the meniscus by the meniscus angle approach as discussed in **Chap. 3**. As this measurement relies on the changes of angle in a line segment along the meniscus edge this measurement might also be useful for detecting when a node passes the same measurement point. The changes in the meniscus are most prominent in the center of the image so the center edge detector measurement can be used. An example is shown in figure

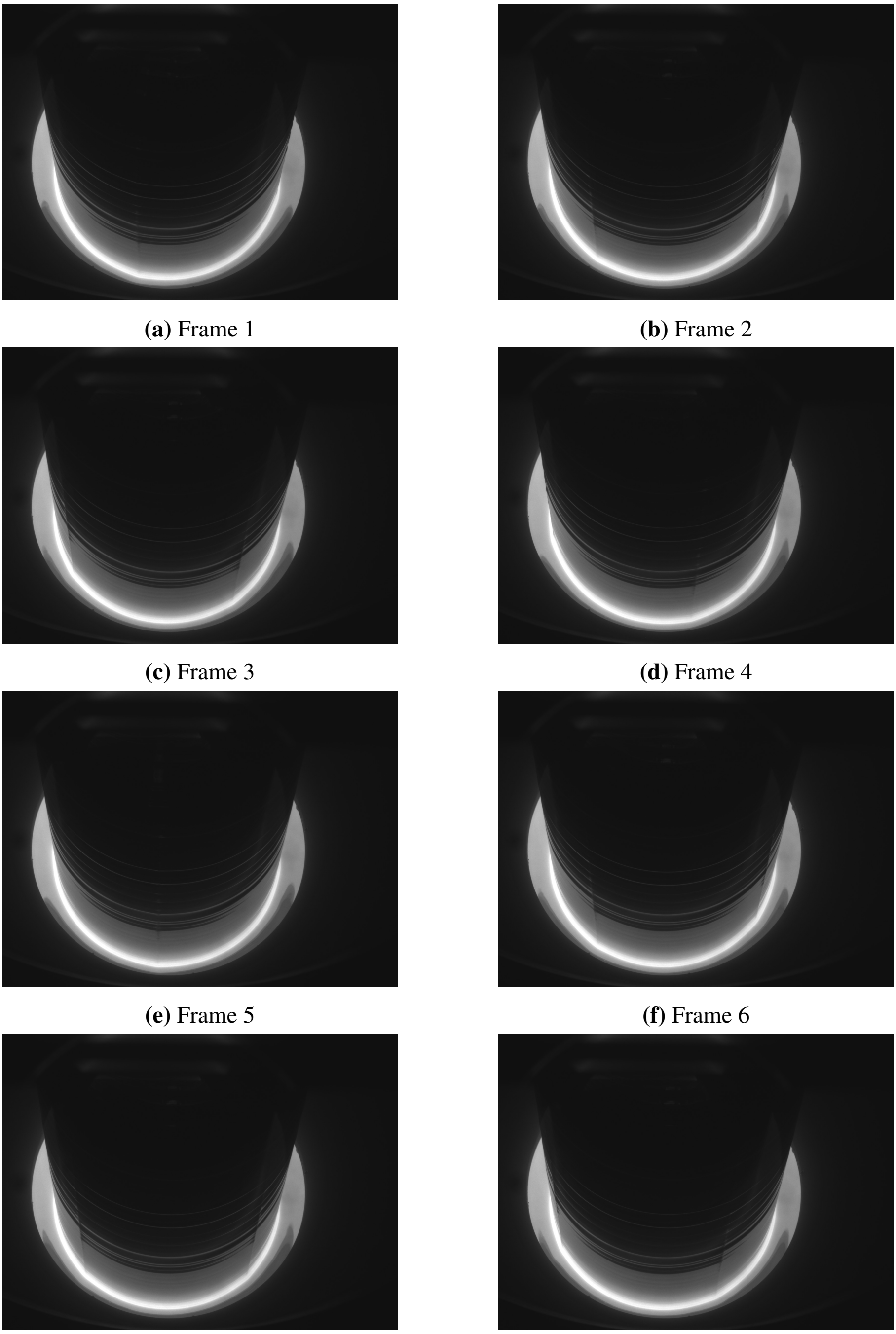
From the previous work described in **Chap. 3** two hypotheses are deduced from the knowledge of how the meniscus angle method works.

Hypothesis one, the peak detection approach: The peaks in the resulting graph as seen in **Fig. 5.6** from cold ingot detection in one of the three edge detectors should represent a node passing the edge detector. If the nodes can be consistently detected then figuring out when one or more of them is missing should be trivial.

Hypothesis two, the mean value approach: The mean value of the resulting graph from the cold ingot detection as seen in **Fig. 5.6** should be lower for an ingot without nodes than an ingot with all nodes intact.

5.2 The peak approach

As the peak approach uses the result from the cold ingot detection algorithm it is important to recognize that the cold ingot detection uses a data set of 30 images to calculate the standard deviation which was deemed appropriate in the previous work on that algorithm. After the first batch of 30 images have been evaluated it discards the oldest image and



(a) Frame 1

(b) Frame 2

(c) Frame 3

(d) Frame 4

(e) Frame 5

(f) Frame 6

(g) Frame 7

(h) Frame 8

Figure 5.3: Eight consecutive frames of rotation of an ingot with all nodes intact.

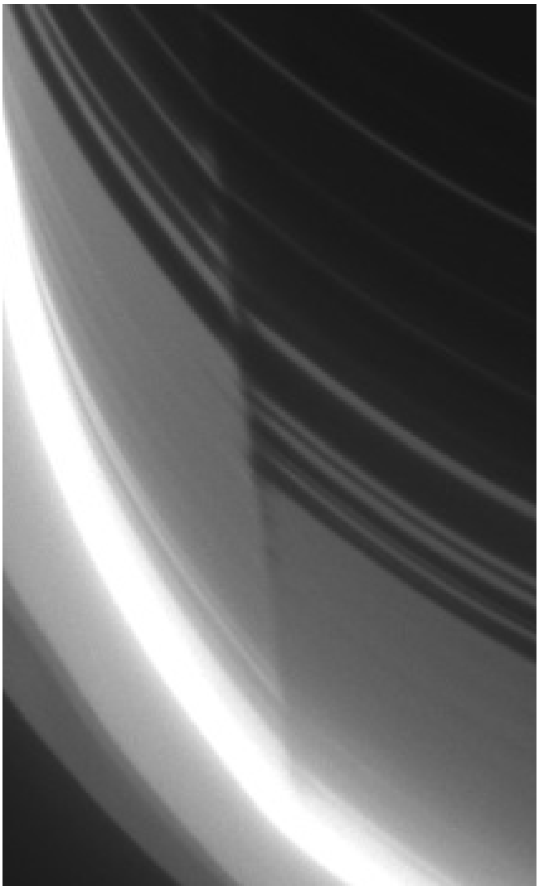
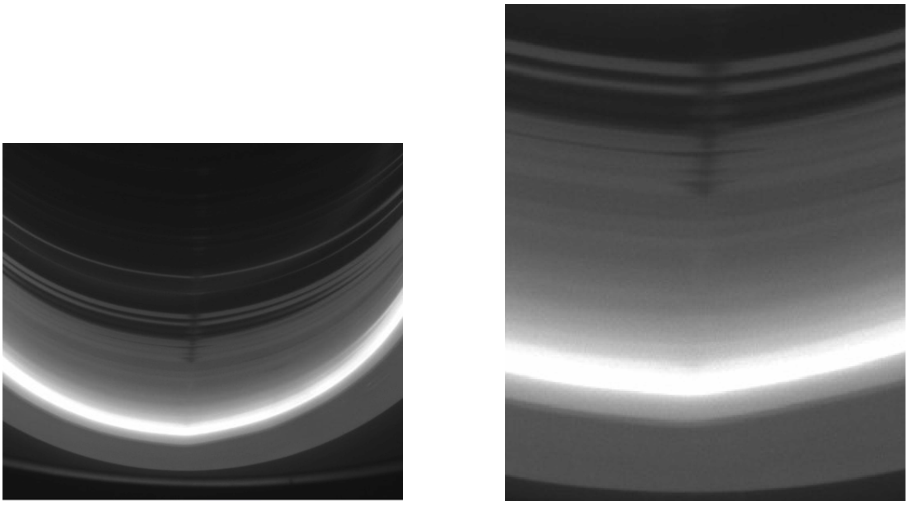


Figure 5.4: A zoomed in view of the left side node from **Fig. 5.3g**.



(a) View of a node passing in the center of the image.

(b) Zoomed in view of a node passing in the center of the image

Figure 5.5: Illustration of geometrical changes in the meniscus as the node passes directly in front of the camera.

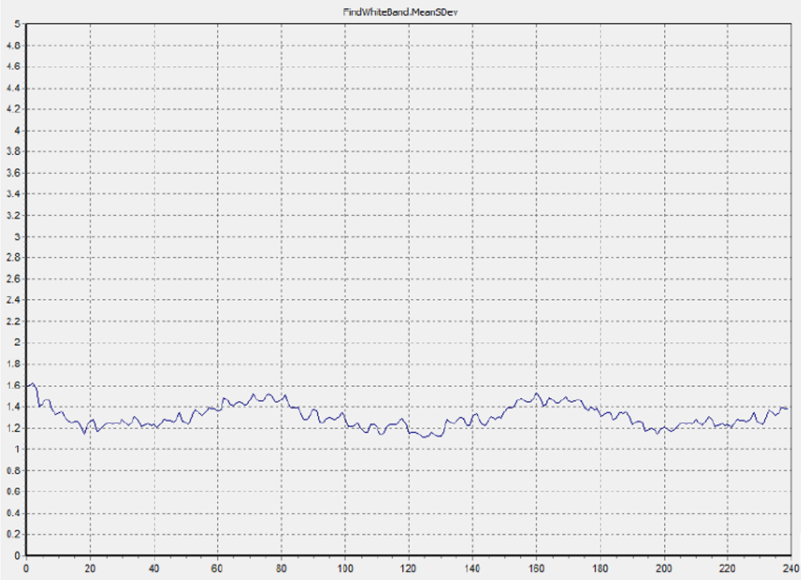
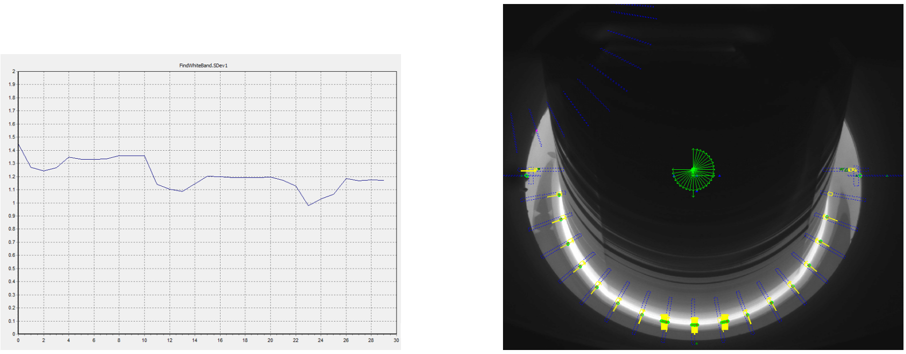


Figure 5.6: An ingot observed over 1 hour and 20 minutes.



(a) Graph of the meniscus angle standard deviation over 30 frames measured in the center edge detector.

(b) The ingot detection setup where the center yellow marked edge detector is the one being evaluated.

Figure 5.7: An example of the standard deviation of the meniscus angle being evaluated.

inserts the newest evaluation to keep the set at 30 observations in a first in first out (*FIFO*) order. So for this approach the cold ingot data from 30 images at a random point during an ingots body phase is extracted and used for further analysis. The extracted data graphed shown in **Fig. 5.7a** and it is observed in real time while looking at the images being evaluated one after another. A trend is observed where local maxima in the graph comes 1-2 frames after the node has passed the detector and also a dip in the graph is observed right before a new node passes. This is illustrated in **Fig. 5.8**.

The peaks noted in **Fig. 5.8** are the local maxima after a node has passed while the dip refers to the local minima at the last frame before the node passes. The peak will usually occur 1-2 frames after the node has passed the detector. This is because the measurement being graphed is the standard deviation of the angle in the edge detector over 30 frames. The node passing will sometimes affect the angle of the meniscus edge for more than one frame which increases the standard deviation and thus becomes a local maxima after two frames have been evaluated instead of only one. Other times the node passing wil only affect the meniscus line angle for one frame and thus the standard deviation will peak the frame after the node has passed. The dips on the other hand have been chosen based on the last frame before the node passes. This produces the phenomena as seen in dip 3 in **Fig. 5.8**. The node passes the frame after the local minima in the dip. This is the same problem as for peaks. Sometimes two frames will affect the meniscus angle and sometimes only one frame will. In the case of dip 3 the last position of the node is close enough to affect the angle of the edge without having passed the measurement point. Wheter to chose the frame before or after the node passes or the closest peak is decided by the approach taken to identify the node.

Observing table **Tab. 5.1** we can see that there are about 11 frames between every peak and dip. This correlation is true also for the passing of nodes. Another observation is that the lowest peak value in the span of 30 images is lower than the highest dip value. This is because the values are so small that small changes in the meniscus or ingot can raise or lower the entire mean value of an entire 30 image dataset, especially if the ingot becomes

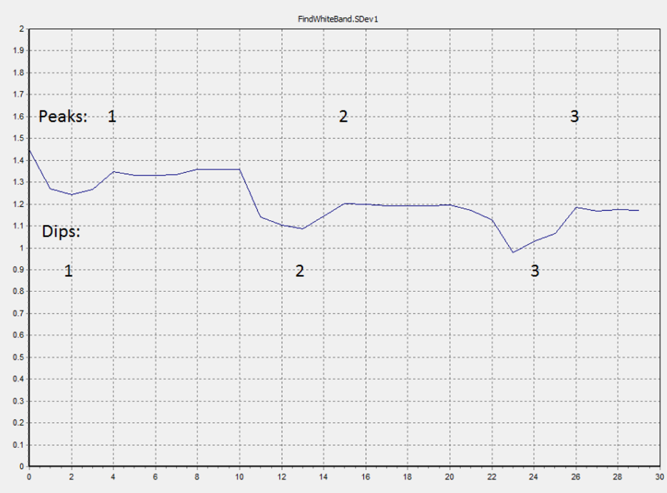


Figure 5.8: Numbered peaks and nodes to indicate when a node passes the evaluated edge detector.

Type	Number	Frame number	SDev Value
Peak	1	4	1.35
Peak	2	15	1.20
Peak	3	26	1.19
Dip	1	2	1.24
Dip	2	13	1.09
Dip	3	24	1.03

Table 5.1: Numeric values of node passes from the graph in **Fig. 5.8** rounded to two decimal places.

slightly cold.

5.3 Derivative peak detection

Since the peaks and dips are clearly correlated with the nodes passing the edge detector where the standard deviation is being calculated, we can try to use a derivative filter to find the nodes. One of the main problems is that there are several smaller peaks and dips which will be reported as false positives with a sensitive derivation filter. What we are looking for is either the dips or the first peak after a dip. A test setup where the standard deviation data is designed for easy testing without integrating the functionality to Scorpion Vision. The derivative approach uses numpy's gradient function to get the derivative of the data set as seen in the **Appendix**. The data set is the same as graphed in **Fig. 5.7a** so the detected peaks should be on the same frames as in **Table. 5.1**.

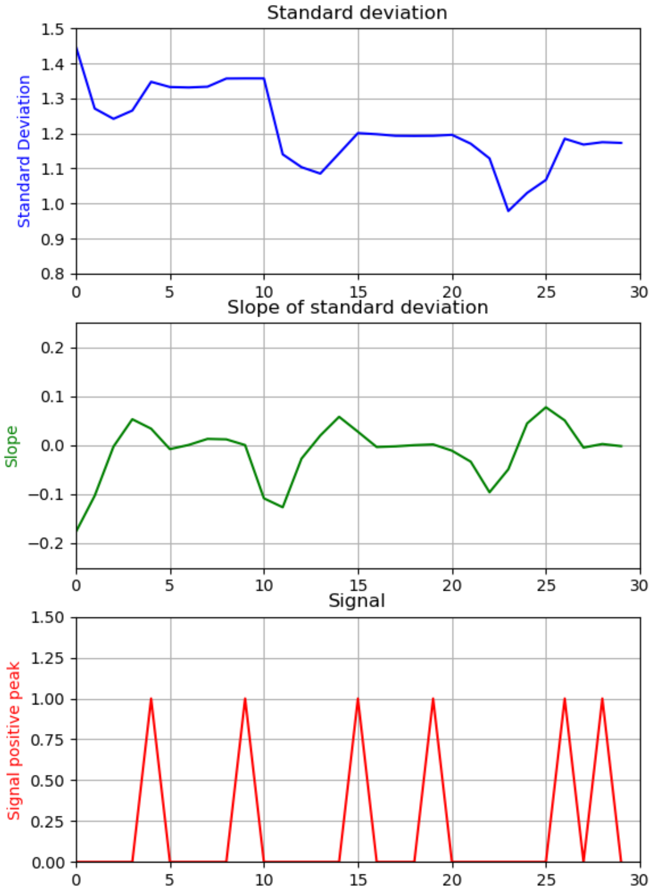


Figure 5.9: The result of a derivative filter attempting to find the positive peaks.

5.3.1 Results

Type	Number	Frame number	SDev Value
Peak	1	4	1.35
Peak	2	9	1.36
Peak	3	15	1.20
Peak	4	19	1.19
Peak	5	26	1.19
Peak	6	28	1.18

Table 5.2: Results of the derivative filter for positive peaks taken from **Fig. 5.9**.

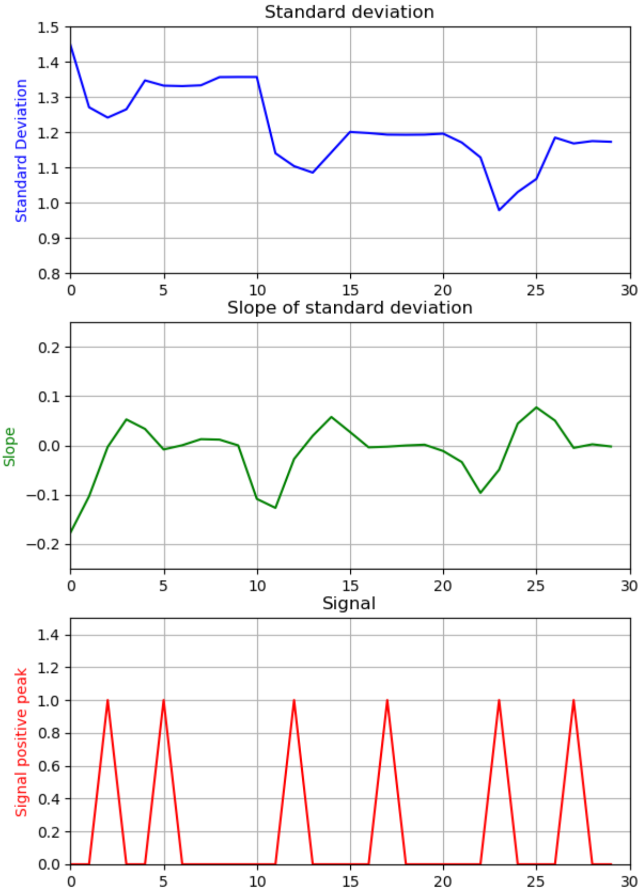


Figure 5.10: The result of a derivative filter attempting to find the dips (negative peaks).

The derivative filter is able to pick up all the peaks from the time series data and correctly place them. As predicted, several peaks which are found do not correlate to a node passing the edge detection as seen when **Table. 5.2** is compared with **Table. 5.1**. This is due to small variations in the ingot and the meniscus when the node is not passing the edge detector which creates small fluctuations and thus unwanted peaks are found. The same phenomenon is seen when graphing the dips for the same data shown in **Fig. 5.10**

So just as the peaks were well detected, every dip is also detected from this data set. There is a clear difference in standard deviation value between the dips and the peaks where the nodes pass the edge detector compared to where they do not as discussed earlier in this chapter. It is also known that the dip occurs before the node passes the edge detector. So one possible solution is to check the difference in value between the previous dip and the current peak whenever one is discovered. Then set a threshold to accept the peak

Type	Number	Frame number	SDev value
Dip	1	2	1.24
Dip	2	5	1.33
Dip	3	12	1.10
Dip	4	17	1.19
Dip	5	23	0.98
Dip	6	27	1.17

Table 5.3: Results of the derivative filter for negative peaks taken from **Fig. 5.10**.

as a node passing the edge detector. For this to be a viable solution there needs to be a guarantee that all the non-node peaks and dips have a smaller difference then when there is no node passing. From **Tab. 5.1** it is known that a node passes between frame 2 and 4, frame 13 and 15 and between frame 24 and 26. So now the difference between peaks and dips where the node passes compared to where there is no node will be compared. For the difference the values and frames where the derivative filter is able to find the peaks and the dips have to be used, so working with values from **Tables. 5.2** and **5.3** we get:

Between frames	Node	Difference
2-4	Yes	0.11
12-15	Yes	0.10
23-26	Yes	0.21
17-19	No	0.00
5-9	No	0.03
27-28	No	0.01

Table 5.4: Difference in peak and dip value on node passes compared to non-node passes.

There seems to be a clear correlation in difference value between a dip and a peak when a node passes versus when a node is not passing. This data set only contains 30 pictures which corresponds to 10 minutes of a run. A full body phase can take up to 10 hours so more data needs to be analyzed.

Some nodes will not land exactly on the edge detector and thus need 2 frames to affect the standard deviation, while other times it will land perfectly on the edge detector and thus affect the standard deviation more in one frame. To combat the issue of node placement while measuring, their passes will be either described by the frame they are on top of the edge detector, or a range of frames from before it hits to edge detector to after.

For this data set the zero crossings and the differences between dip and closest peak have been calculated the same way as done for **Table. 5.4**. The derivative filter returns results as in **Table. 5.5**.

The first thing to notice is that the data set has peaks and dips at a higher interval than the previous dataset. This frequency is dependent on the rotations per minute and the ingot diameter which can change between runs and throughout a run on the live environment.

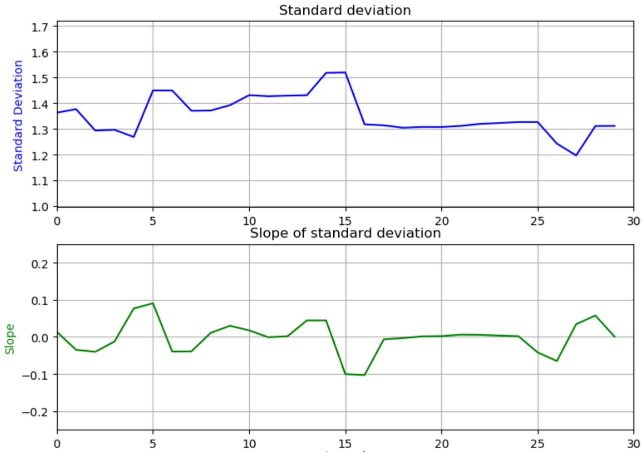


Figure 5.11: A dataset on a different ingot in the body phase.

Type	Number	Frame	SDev Value
Peak	1	0	1.36
Dip	1	3	1.30
Peak	2	5	1.45
Dip	2	7	1.37
Peak	3	10	1.43
Dip	3	11	1.42
Peak	4	14	1.52
Dip	4	18	1.30
Peak	5	24	1.33
Dip	6	26	1.24

Table 5.5: Zero crossings by numerical values in data set 2.

Frame	Node
0-1	Yes
5	Yes
9-10	Yes
14-15	Yes
19	Yes
23-24	Yes
28	Yes

Table 5.6: The frame where the nodes actually pass or hit the edge detector in data set 2

Since the data set only contains an image every 20 seconds the frequency of nodes depends a lot on these factors. The second thing to notice is that this data set begins with a peak and ends with a dip. The approach is to look at the difference between a dip and the consecutive peak. Thus the first peak has to be discarded and the last dip has to be discarded. Thus the differences between relevant dips and their consecutive peaks are as shown in **Table. 5.7**.

Frame	Node	Difference
3-5	Yes	0.15
7-10	Yes	0.06
11-14	No	0.09
18-24	Yes(*)	0.02

Table 5.7: The difference between dips and consecutive peaks for data set 2. (*) For this range there is supposed to be 2 nodes passing, one at frame 19 and one between frame 23 and 24.

5.3.2 Discussion

As **Table. 5.7** shows that the derivative filter approach has flaws. To avoid overlooking any nodes at the beginning or the end of a data set it needs to begin with a dip and end with a peak. This can be remedied by continuously updating the data set frame by frame instead of 30 by 30 frames. This might introduce another problem as the gradient might have problems handling the edges as seen in **Fig. 5.11** it finds a false peak at frame 0, which is supposed to be at frame 1. The differences between a dip and a consecutive node is at the minimum 0.06 in the range where a node is found for data set 2. In data set 1 the highest difference between a dip and a consecutive node is 0.03 which has a low margin of error if a constant threshold is to be set. This can be handled by setting an adaptive threshold but it is difficult to decide what factors should decide this threshold. There is also a big question of how big difference a 5 frames per second update frequency in the live setup would change the dynamic of the derivative filter compared to the 20 seconds per frame update on the test set up. The derivative approach might work better on a smaller data set than that which is extracted from 30 images. It is difficult to set this horizon statically and do any tests on a smaller set since in a smaller set the differences resulting from the low update frequency on the test setup will have a bigger effect than the general trend seen by a larger data set. This approach is able to show a general trend but it is not robust enough to consistently detect nodes throughout a run. With a reduction of the window in which to look for local maxima and minima the algorithm might perform better, but the window size and analysis of performance should be done on the live setup to guarantee satisfactory performance. If there is a guaranteed amount of frames between nodes in the live setup then the window in which local maxima and minima are detected to guarantee only one node passing and then look for a global maxima within that window. The size of this maxima could be used to separate existing nodes from melt fluctuations.

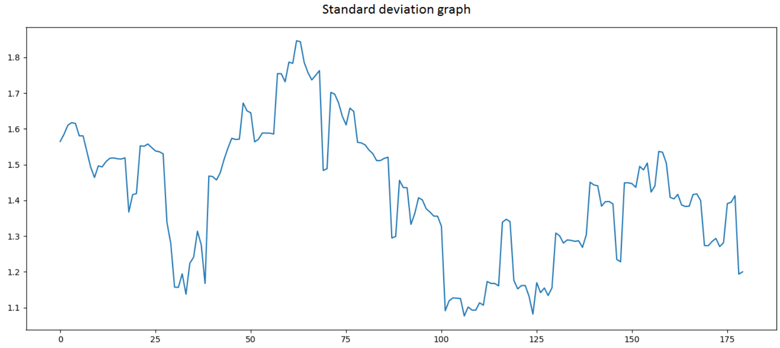


Figure 5.12: Recurring standard deviation phenomena at a specific frequency over a period of 1 hour.

5.4 Frequency analysis peak detection

The standard deviation seems to oscillate with small peaks indicating nodes throughout a run. The peaks appear at a quite constant interval as shown in **Fig. 5.12**. In this section the fourier transform and the wavelet transform will be used to detect the peaks or find an underlying characteristic that can be used for node detection. Both the continuous wavelet transform and fourier transform will be performed using their built-in functions in the SciPy library. The short time fourier transform is used as an attempt to quantify the change of the nonstationary signal's frequency over time.

5.4.1 Results

The fourier transform and the wavelet transform are both run on the same set of data which is seen in **Fig. 5.12**

The Fourier transform response is seen in **Fig. 5.13**. The response shows the peaks by a brighter colour over the evaluated frames which are shown on the x axis. The Y axis shows difference in characteristics over the frequencies evaluated within the window.

From **Fig. 5.13** it seems that every new frame on frequency 0.1 contains the most of the data sets characteristics. It is unable to pick up any information happening at specific frequencies thus making the approach unviable for the sake of detecting peaks in the data set.

The continuous wavelet transform is able to use the frequency response to detect most of the peaks in the signal as shown in **Fig. 5.14**.

The wavelet transform seem to have the same problem as the derivative filter approach. many of the peaks are detected, but there are also peaks lacking. A width parameter for the peak detection with continuous wavelet transform can be changed to the size of expected peaks. Tuning this value gives slightly different results as seen in **Fig. 5.15**. But for both graphs there are important peaks missing, such as at frame 16, 68 and 85. Plus several detected peaks are not correctly detected.

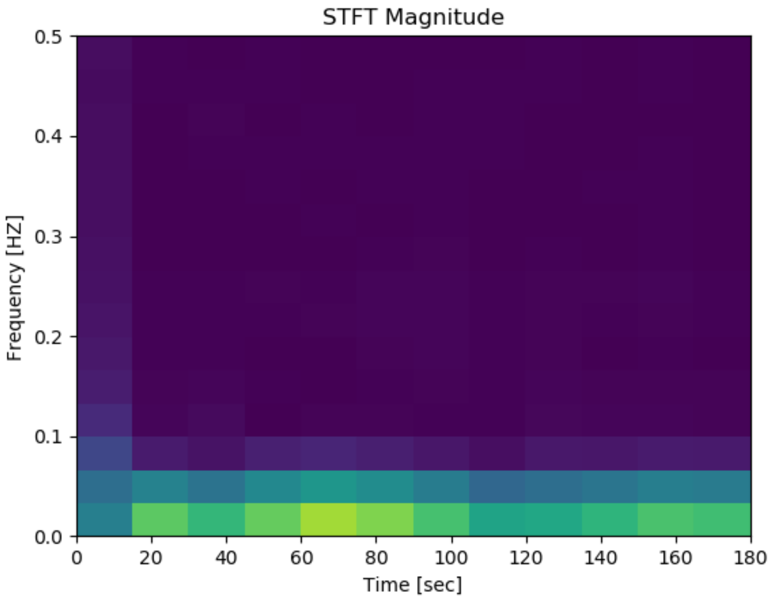


Figure 5.13: The short time fourier transform response trying to detect frequency characteristics for the data set.

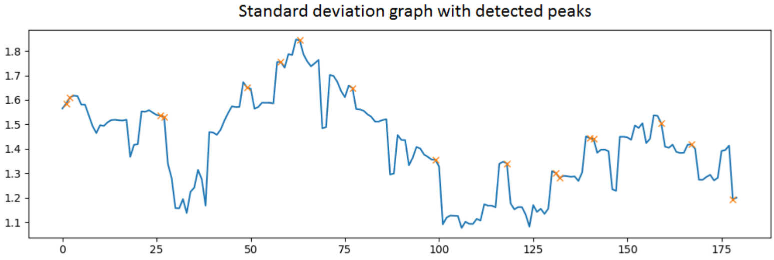


Figure 5.14: Peak detection from the continuous wavelet transform indicated by orange crosses on the same data set. Expected peak width set to $1-3$ frames

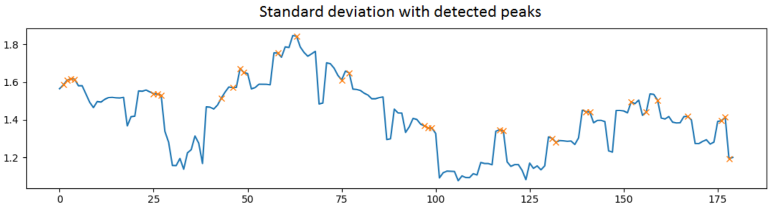


Figure 5.15: Peak detection from the continuous wavelet transform indicated by orange crosses. Expected peak width set to $1-8$ frames.

5.4.2 Discussion

From the two proposed techniques the continuous wavelet transform seems to be the most promising one. Although the wavelet transform performed quite well on the data set, it seems to have similar problems as the derivative approach where not all peaks are properly detected and some peaks are not detected at all. By tweaking the wavelet transform around the desired result from the graph its performance can be increased. The problem with this approach is that during a crystal growth phase of an ingot, the standard deviation of the meniscus angle may wary a lot due to physical variations. This makes the data set too volatile to make tuning the wavelet transform specifically for the dataset a viable method of detection.

5.5 Smoothed Z-score peak detection

Z-score, or the standard score is a statistical technique to find out how many standard deviations a value is from a mean. In the case of structure loss this can be applied to the graph of the mean standard deviation of the line segment angle to look for any big value changes.

The Z-Score formula is as follows:

$$z = \frac{x - \mu}{\sigma} \tag{5.1}$$

where z is the z-score, x is the evaluated data point, μ is the mean of the data set and σ is the standard deviation.

A problem is that within a set of 30 evaluated images, the dips in one spot might have a similar value to a peak in another spot as discussed in **Sec. 5.3**. Thus the Z-score approach needs some extra features. The algorithm, which is shown in the **Appendix**, takes three inputs. Lag, Threshold and Influence. Lag is the moving window for smoothing the data. Threshold is the amount of standard deviations before signalling a peak while influence is the factor of which new data should be used to recalculate a new threshold from the initial input. Tweaking of these factors might make the Z-score approach feasible for finding the peaks of the graph result from the cold ingot detection.

5.5.1 Results

The algorithm used for the Z-score implementation is a Python implementation of pseudo code found on *Stackoverflow*. Brakel (2016)

For the first example the initial conditions are $Lag = 5$, $Threshold = 3.5$ and $Influence = 0.5$. The light blue line is the moving mean, the green is the adaptive threshold calculated by initial threshold combined with the influence of new data and the dark blue line is the actual data set. The red lines indicate positive signalling or negative signalling if a data point is found above or under the threshold.

Fig. 5.16 shows that the threshold is way too large to detect most of the peaks and dips as well as the moving mean changing too slowly to pick up the dynamics of the system. This indicates a too big threshold and influence ratio and the lag needs to be turned down as well. After tuning the three input variables the best result on this data set is achieved by

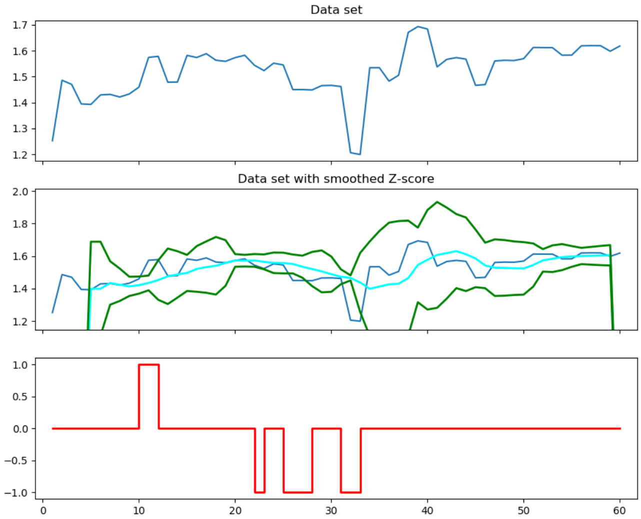


Figure 5.16: Initial Z-score with $Lag = 5$, $Threshold = 3.5$ and $Influence = 0.5$.

setting $Lag = 2$, $Threshold = 1$, $Influence = 0.7$ which results in the graph shown in **Fig. 5.17**.

5.5.2 Discussion

When the Z-score is run with well chosen input values for the data set it is evaluating, it returns a good result where almost all the dips and peaks are found. Notably in the set graphed in **Fig. 5.17** is the absence of any peak at the 24 frame mark as well as the 44 frame mark. The dip between frame 35 and 38 is also missing. The good performance is a result of spending time specifically tuning the input to get as many of the dips and peaks registered as possible for this specific data set. With a different data set the variations will differ and the chosen initial conditions will not necessarily give the best performance. Hence this method of finding peaks may not be consistent enough to detect peaks in a given data set.

5.6 The mean value approach

The mean value approach is based off the second hypothesis which theorizes a detection by looking at how the mean value of the standard deviation will decrease as the nodes disappear. The edge detectors on the meniscus evaluate how their line segment angles change. As the nodes disappear the ingot and thus the meniscus will become a pure ellipse and the angle of a line segment along the meniscus should barely change, if at all. The method of evaluating the viability of the mean value approach does not require any new algorithm to be designed since it relies on the same geometry as the *ingot_squareness* variable which

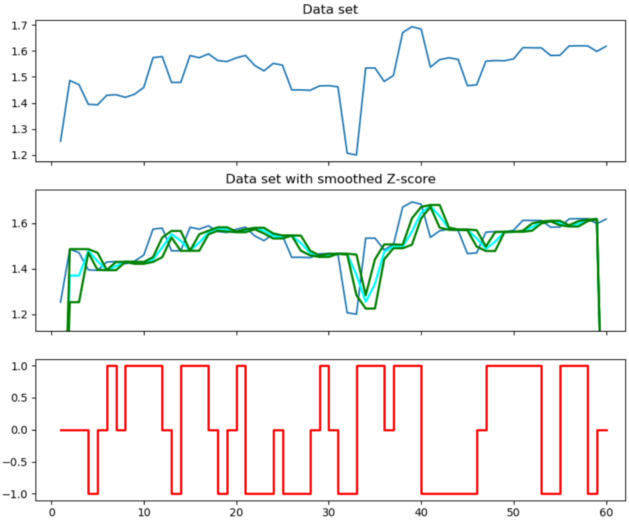


Figure 5.17: Better Z-score with $Lag = 2, Threshold = 1$ and $Influence = 0.7$

is already implemented. This approach will simply be an analysis and comparison of the time series graph of the *ingot_squareness* value when an ingot is experiencing structure loss and when it is healthy.

5.6.1 Results

To see if the mean value approach has any merits, the mean standard deviation of the line segment angles along the meniscus are analyzed to look for trends in data sets where there is structure loss in the ingot. Two such sets are shown in **Fig. 5.18** and **Fig. 5.19**.

There are two big factors which affect the standard deviation graph when the ingot loses structure. The mean value seems to fall to around 1.0 and the small peaks indicating node passes are also noticeably smaller. The value reduction of the mean standard deviation is quite large and might in itself make for a good detection of ingots experiencing structural loss. To find a potential threshold between wanted state and structure loss, an ingot with all four nodes intact needs to be analyzed. This is to ensure that an ingot with all four nodes intact never reaches a mean standard deviation of close to 1.0. If only one ingot with all four nodes intact reach a mean standard deviation of 1.0 then there is no certainty it will not happen with other ingots and the method will not be robust enough. The graph in **Fig. 5.20** shows that a healthy ingot with all four nodes intact can have a mean standard deviation as low as almost 1.1. This means that the margin of error is very small.

Since there is a difference in dynamics between the live setup and the test setup in regards to number of frames being evaluated per second there might be a difference in how much a structure loss wil affect the standard deviation. So some data from the live factory setup is gathered to see if the difference is more easily spotted. As seen in **Fig.**

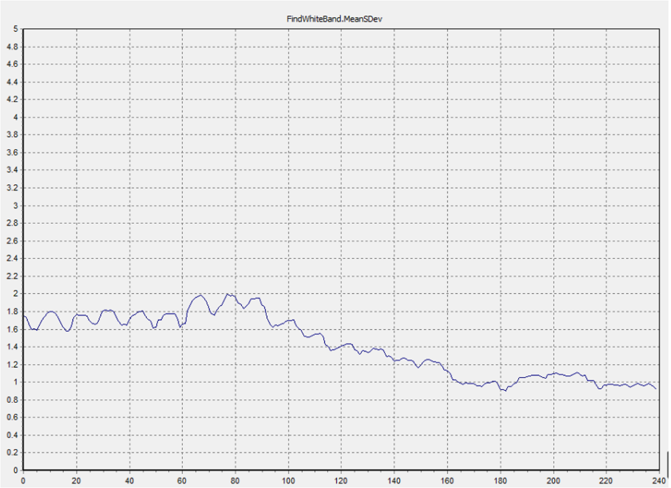


Figure 5.18: A simulated graph of an ingot in the time frame where it loses structure.

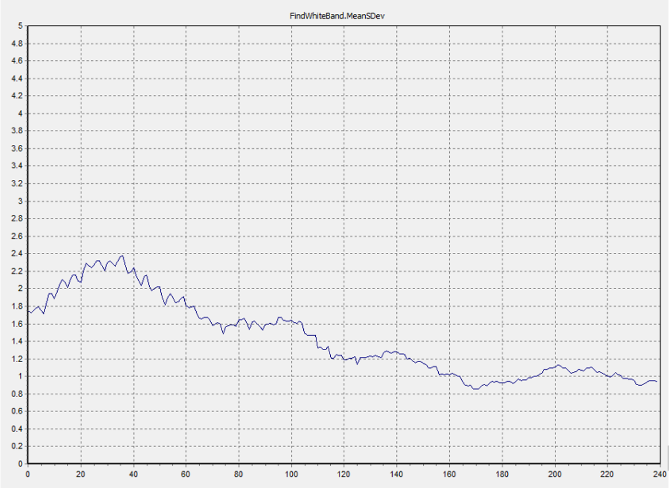


Figure 5.19: A simulated graph of another ingot in the time frame where it loses structure.

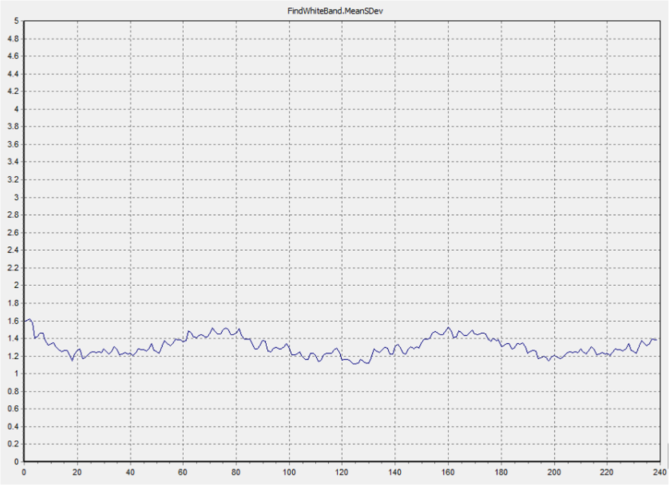


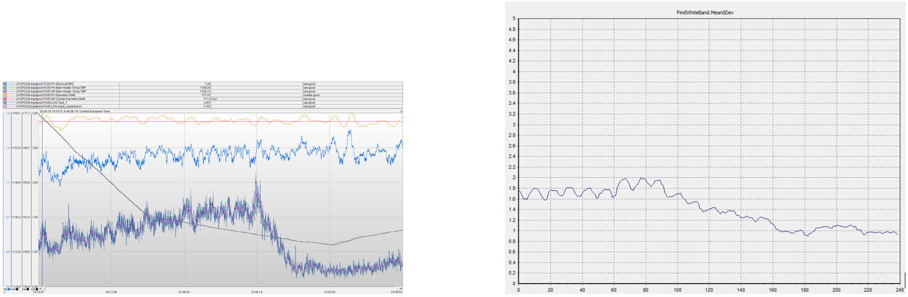
Figure 5.20: A simulated graph of an ingot in a healthy state with all four nodes intact.

5.21 and **Fig. 5.22** the *APIS* plots from the live setup show a much more drastic change in value when the ingot loses structure. The *ingot_squareness* variable goes as low as 0.6 at some points.

5.6.2 Discussion

Due to the differences in frequency of updates between the live setup and the test setup, every effect changing the mean standard deviation value will have a larger impact on the live setup than on the test setup since more frames will report the bigger changes thus affecting the standard deviation more heavily. This might be enough of a difference to set a constant threshold to separate a healthy ingot with four nodes intact to an ingot which is experiencing structural loss. For this to become a reality, the *APIS* charts for a large amount of ingots have to be checked for their minimum *ingot_squareness* value while being healthy compared to the maximum *ingot_squareness* value of an ingot experiencing structural loss.

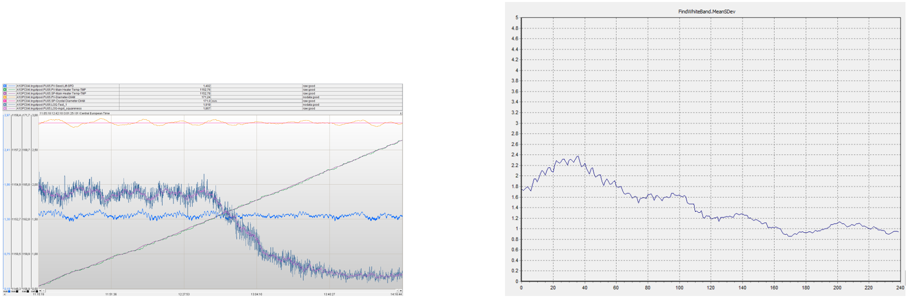
Even if a constant threshold is not achievable, an approach involving the slope might be possible. By looking at the negative slope of the curve and make a connection between how long or fast the curve is declining and connect that to the loss of structure in an ingot. This approach however needs to make sure that the decline in *ingot_squareness* is not a result of an operator manually increasing the temperature on a slightly cold ingot. As seen in **Fig. 4.9** the characteristics of a cold ingot with manually increased temperature results in a similar descent of the *ingot_squareness* variable as structural loss. This can be compensated for since the *PLC* can record when the temperature was adjusted and start a timer to block any alarm indicating a structure loss for a period of time.



(a) *APIS* graph of the *ingot_squareness* variable decreasing upon a structure loss

(b) A simulated graph of the same ingot in the time frame between to 14:00 on the test setup

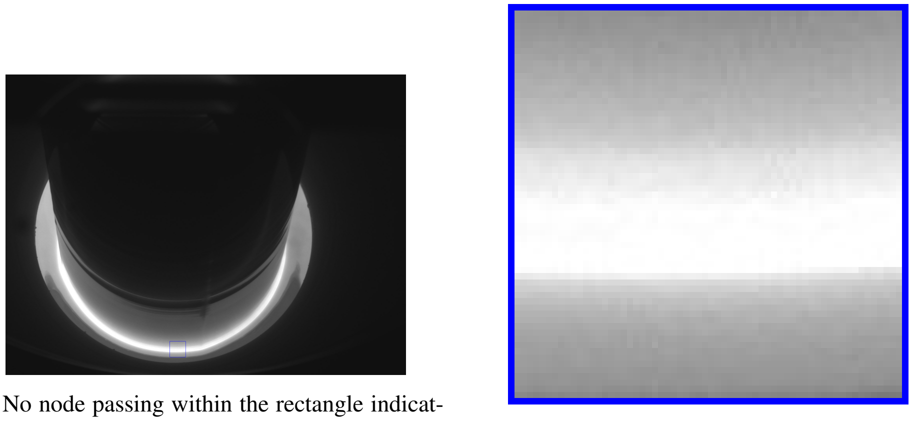
Figure 5.21: Real time and simulated standard deviation value comparison on an ingot experiencing structure loss the 30th of April 2018.



(a) *APIS* graph of the *ingot_squareness* variable decreasing upon a structure loss

(b) A simulated graph of the same ingot in the time frame between to 14:00 on the test setup

Figure 5.22: Real time and simulated standard deviation value comparison on an ingot experiencing structure loss the 11th of May 2018.



(a) No node passing within the rectangle indicating the *ROI*

(b) The cropped image within the *ROI*

Figure 5.23: The position of the rectangle and the cropped image showing no node within the *ROI*.

5.7 Machine learning approach

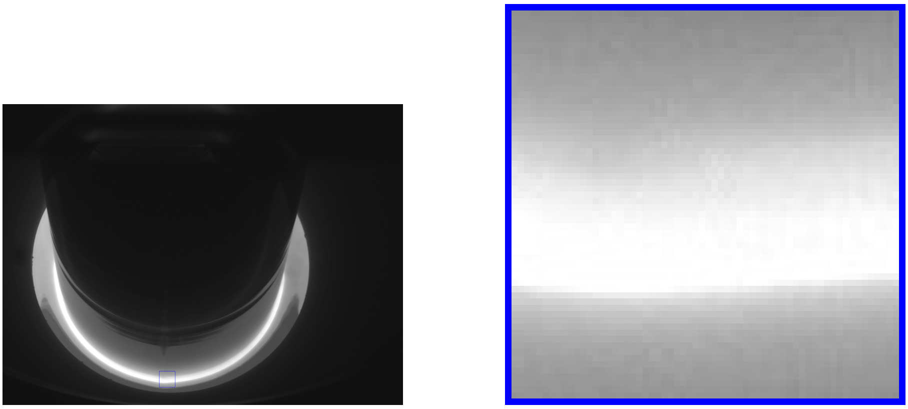
This approach is not exactly hypothesized in the chapter introduction but during the work on the other methods it seems like a natural way of detecting something so easy to see with the human eye and so elusive for other methods. If a machine learning approach can be trained to detect images with nodes, then an already trained machine learning model can run on the puller checking every new frame for a node within a cropped area. If it is consistently able to detect nodes, then the logic to look for a missing node should be trivial. In this regard the machine learning approach is quite similar to the peak detection approaches. Both the code for the training script and the prediction script by loading the model can be found in the **Appendix**.

For fast detection, the only area which needs to be evaluated is a *ROI* of the meniscus where either a node passes or does not pass. Thus a small portion of the image can be cropped to speed up the training process of the model. The interesting area and their respective crops with and without nodes are shown in **Fig. 5.23** and **Fig. 5.24** The cropped images for this approach have a size of 64x64px but potentially their size could be even smaller.

As seen in **Fig. 5.24b** the geomtrical changes that should be detected are miniscule.

5.7.1 Method

An artifical neural network is designed by using the *Keras* and *TensorFlow* libraries described in **Sec. 2.3.8** and **Sec. 2.3.7**. The neural network is a simple network with a low amount of neurons to avoid finding complex correlations that are not relevant for the detection of a node. It has 32 neurons in the first layer, 16 neurons in the second layer then it becomes fully connected in the third layer with a sigmoid activation to return a value between 0 and 1. This result translates to the model classifying the images as with node, or without a node. The model is compiled with an optimizer from the *Keras* library called



(a) A node passing within the rectangle indicating the *ROI*.

(b) The cropped image of the *ROI* containing a node.

Figure 5.24: The position of the rectangle and the cropped image showing a node within the *ROI*.

nadam and it runs a binary crossentropy loss function that should be minimized. There are dropouts introduced at every layer of the model to avoid overfitting as discussed in **Sec. 2.1.2**.

Images from several hours from three different ingots are cropped and then manually classified into node or no node images. There is an abudance of images showing no nodes so many of these no node images are removed to ensure the model is not underfitted for node detection. The cropping of the images is done by a script to ensure correctly sized regions are cropped from a large data set of images. When image sets from different ingots are cropped, their position might be slightly different compared to each other. For this approach the position of the crop has been manually placed which may introduce problems later. Ideally the crop should always be performed in the same position by dynamically finding the *ROI* to crop from, as for example the center meniscus edge detector discussed in **Chap. 4**.

The training set contains two classes of images, nodes and no nodes. In total there are 140 images in the training set and 32 images in the validation set. The training set contains 44 images with nodes and 96 without nodes. The model then validates itself on a validation set which also contains the same two classes of images. The validation set contains 16 images of each class. The images in the validation set should be images the model has not seen while training. This means that a high validation score corresponds to detection of the phenomena in images in has not seen before.

This method was quickly designed for conceptualizing the approach so the method is highly based upon trial-and-failure. The training of the data set is done over 50 training cycles, or *epochs*. The amount of epochs to train depend on the individual data set. Some data sets converge faster than others depending on their characteristics.

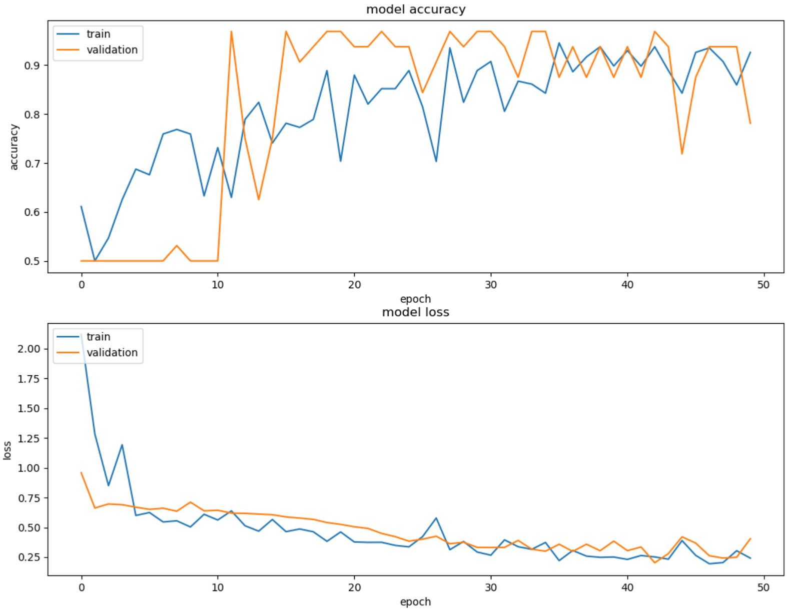


Figure 5.25: A plot of training and validation scores and loss for a training session over 50 epochs.

5.7.2 Results

The model training in **Fig. 5.25** shows a long period of time between epoch 0 and epoch 10, where the validation accuracy stays at 0.5 even if the training accuracy went up to almost 0.75. This might be due to the training data containing 96 images of no nodes and only 32 images of nodes. Thus the first 10 epochs the model was only able to learn how to detect an ingot with no node present and therefore only the images with no nodes from the validation set were classified correctly. During the 50 epoch training period the training accuracy seems to grow steadily until about epoch 35 where it only has small fluctuations and also has a relatively even value with the validation accuracy. They are both fluctuating around 0.88 and 0.93 in accuracy at this point. This means that the model can explain which class an image belongs to with about 90 percent accuracy at this point, which is proven by the accuracy value of the validation set fluctuating around the same point.

Since there are 32 validation images and 96 training images it means that the fluctuations of accuracy score differs from the fluctuations of the validity score. One image misclassified in the validation corresponds to a 0.3125 decrease in accuracy. One misclassified image in the training corresponds to a training accuracy decrease by 0.0714.

The loss graph seen in **Fig. 5.25** is a scalar value that should be minimized for best performance. The loss function is a result of binary crossentropy and helps indicate the learning progress of the model. In this case the loss seems to continue decreasing all the way to the 50th epoch, potentially indicating that several more epochs of training may increase the performance.

So the performance is pretty good with a very quickly designed neural network, a

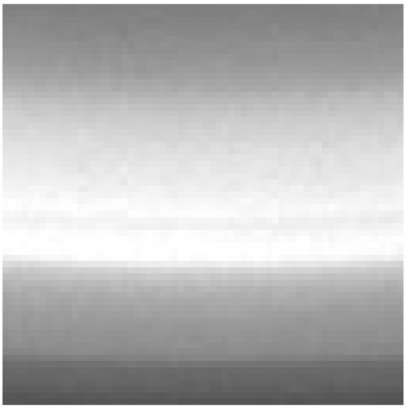


Figure 5.26: Image from an ingot cropped slightly lower than the other ingot crops in the data set.

small data set and non-fixed positions of cropping the ingot. Some of the inaccuracy in the training set may come from a subset of figures that were cropped at a slightly different height than the other ingots as shown in **Fig. 5.26**. There are 18 images from the badly cropped ingot data set in the training set. 12 are without nodes and 6 are with nodes. In the validation set there are no images from this ingot.

These images might be a problem for the training set accuracy as it has problems understanding the black line at the bottom of the image. There are not many images from this data set to train on so the model might misunderstand the difference as more important than it really is.

The results in validation and training accuracy may change every time the model has been run. A model was achieved where the training accuracy was 93% and the validation accuracy was 96% on the same data which is plotted in **Fig. 5.25**. This model is used to predict nodes or no nodes on a data set from a completely unseen ingot from the model perspective. No images from this ingot was used in either training or validation.

The prediction data set contains 79 images from a single ingot. 8 images are with nodes, 71 are without nodes. Out of the 79 images the model correctly predicts 72 of them. All 71 images without a node are classified as images with no nodes and only one out of the 8 images with a node is classified as a node image.

5.7.3 Discussion

The machine learning approach showed promise while validating data sets on the same ingot as it was trained on but failed once it tried to classify images from a new ingot. This is not necessarily an indicator of a bad approach to node detection and detection for structure loss as it might seem. The data set used for training has problems which creates a lot of uncertain factors for this method. The training set is very small for the model to achieve a robust understanding of the problem with only 140 images from three different ingots. Every ingot has specific traits such as rings along the edge of the ingot with different depths and widths. Some ingots have more pronounced nodes while some have more vague nodes. With a larger data set from more ingots, the model should be able to to train itself to handle these differences. A huge factor is that the *ROI* for cropping is placed manually in the script that handles the cropping. If the images being predicted are placed in a different position than the images from the training set, it can create miniscule differences such as the gray region below the meniscus can be larger or have a slightly different angle. These differences are enough for the model to make errors when trying to classify the prediction images. The validation test with unseen images from the same ingots as the training set should indicate that a model trained on a varied data set, with the *ROI* placed in the exact same position relative to every ingot, should produce a highly efficient classification model for node detection.

Chapter 6

Automatic neck temperature detection concept

This chapter will cover an approach for automatically detecting the correct temperature in the neck phase. This phase is very time consuming as the process is very temperature sensitive at this point. This sensitivity combined with temperature fluctuations in the melt makes the task of stabilizing the temperature correctly both difficult and time consuming. This temperature stabilization may take from two hours ranging up to twelve hours depending on the experience of the operator as well as the initial conditions in the puller. This approach does not necessarily directly measure the tmeperature but in stead focuses on measuring the states of the neck as a result of the temperature fluctuations.

6.1 Problem specification

The detection problem is complex and there are several conditions that has to be met to ensure the temperature is correct. These conditions are loosely based upon operator experience and thus they are hard to specify in quantified terms. To explain the conditions, a visualization of the desired state and the possible unwanted states is shown in **Figs. 6.1 - 6.5**.

The unwanted states are: A gray ring in the meniscus seen in **Fig. 6.1** which indicates a too hot melt temperature. Split nodes on the edge of the meniscus as seen in **Fig. 6.2**. A thicker meniscus than desired as seen in **Fig. 6.3** which indicates a too hot melt temperature. A meniscus with unclear edges seemingly blending into the surrounding melt visualized in **Fig. 6.4**.

The desired state is a meniscus without any of the unwanted phenomena and with all four nodes intact around the meniscus. An image form a desired state is shown in **Fig. 6.5**.

Due to the fluctuations of the melt it is not enough to simply be at the desired state for a few image frames before starting the next phase. The neck might be in a good state during a part of the cyclic temperature fluctuations and in a bad state for the rest. To go to the

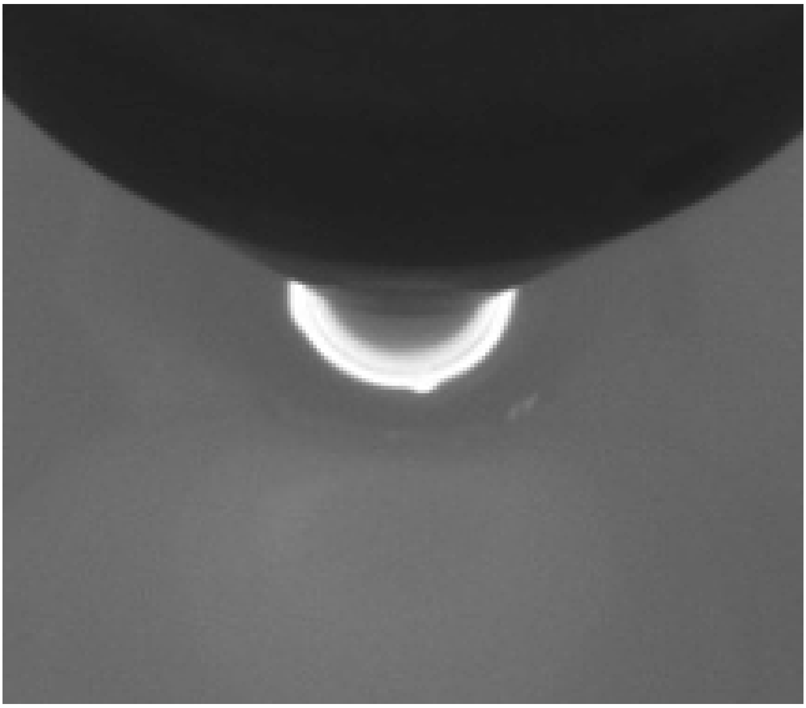


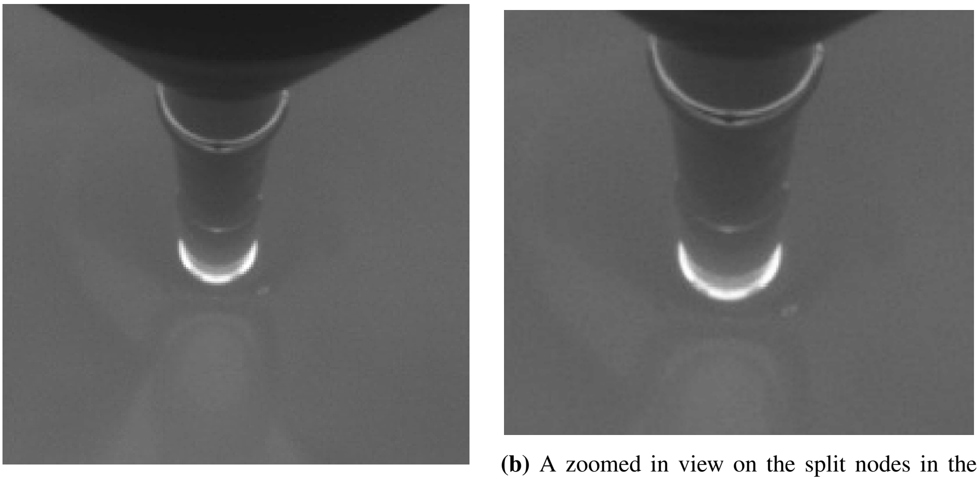
Figure 6.1: Visualization of a neck with a thin gray line in its meniscus indicating that the melt is too hot.

next phase the neck has to be stable in the desired state for a whole cycle of fluctuations. Currently the rule of thumb for the factory operators is a 20 minute period without any changes into unwanted states. This 20 minute period might be longer than necessary, but it is a good guarantee for desired initial conditions in the melt for future growth.

6.2 Method

Before deciding on a specific method to solve the detection problem another question needs to be determined first. If the detection algorithm should simply give a result which indicates a good state or not a good state, or give an indicator as to what is wrong if it returns an unwanted state. In this case the second option could return an indicator if the temperature should be adjusted up or down. This would help guide the factory operator in the process of stabilizing the melt temperature. If the detection algorithm simply returns a binary result it would help the operators by letting them do other work while waiting for the result after a temperature adjustment.

For a binary result the algorithm would only need to detect the desired state and anything else would be considered an unwanted state and indicate an alarm. For the other approach a detection for every unwanted state would have to be implemented and give results depending on which unwanted state is detected.



(a) Visualization of split nodes in the meniscus.

(b) A zoomed in view on the split nodes in the meniscus.

Figure 6.2: View of a meniscus with a split node with and without zoom.

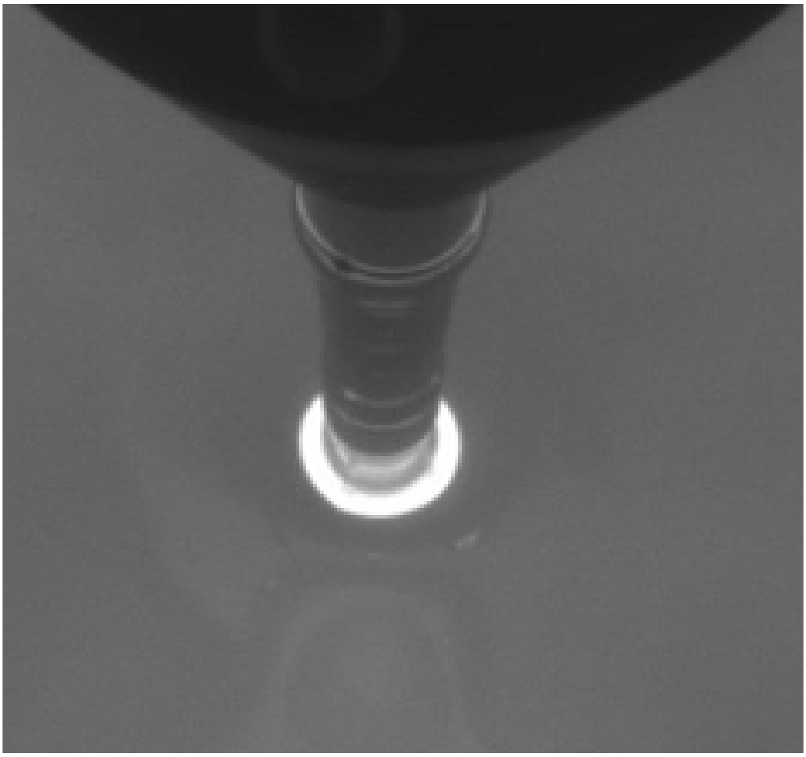


Figure 6.3: Image of a too thick meniscus in the neck phase.

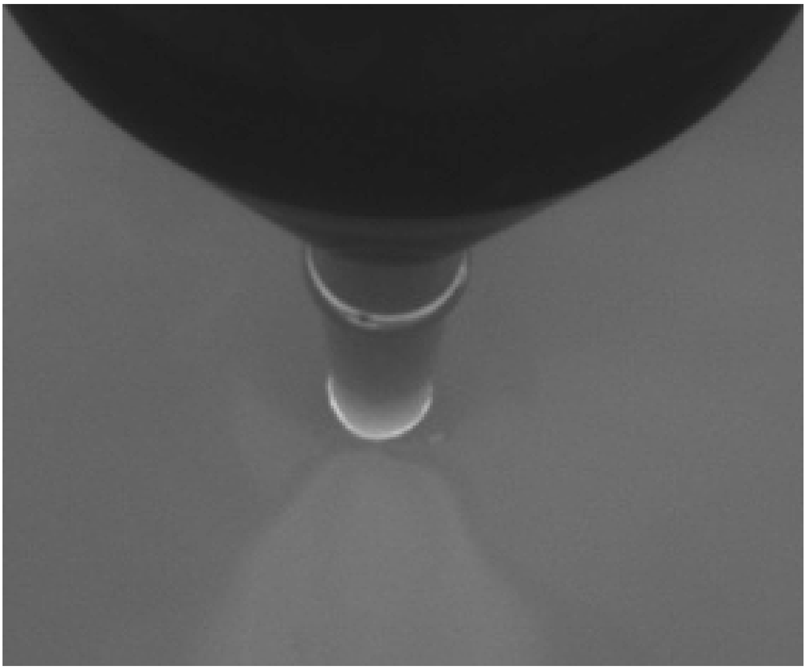


Figure 6.4: Image of a meniscus without a clear edge to the surrounding melt.

Both approaches, either binary result or classified results, need to look at a lot of the same phenomena so many of the same computer vision techniques can be used to detect them.

This chapter will focus on the classified results as this would give an added efficiency for the factory operators compared to the binary result.

A classifiying algorithm needs to detect all the unwanted phenomena from **Figs. 6.1 - 6.4**. So handling each of these states as a separate detection problem and then returning a binary flag to a logic controller which decides what course of action should be recommended to the operator. One way of doing this would be to place edge detectors around the meniscus and look for their shape change in much the same way as the structure loss detection from **Chap. 5**. The structure loss detection approach could be used to find the nodes from **Fig. 6.5** and the split nodes from **Fig. 6.2**. Another edge detector could be placed on the inside of the meniscus which in combination with the outer edge detectors could check the thickness of the meniscus for the problem illustrated in **Fig. 6.3**. If the outer edge detectors fail it would be a sign of an unclear edge as shown in **Fig. 6.4**. The gray ring in the meniscus, as shown in **Fig. 6.1**, could be detected by either trying to place an edge detector inside the meniscus to attach to the gray ring when found, or check for light intensity changes within the meniscus.

Another approach would be to train a machine learning model with a manually classified data set from experienced operators. The machine learning model would need to be trained on a large data set on all the different types of states. The area it needs to train and

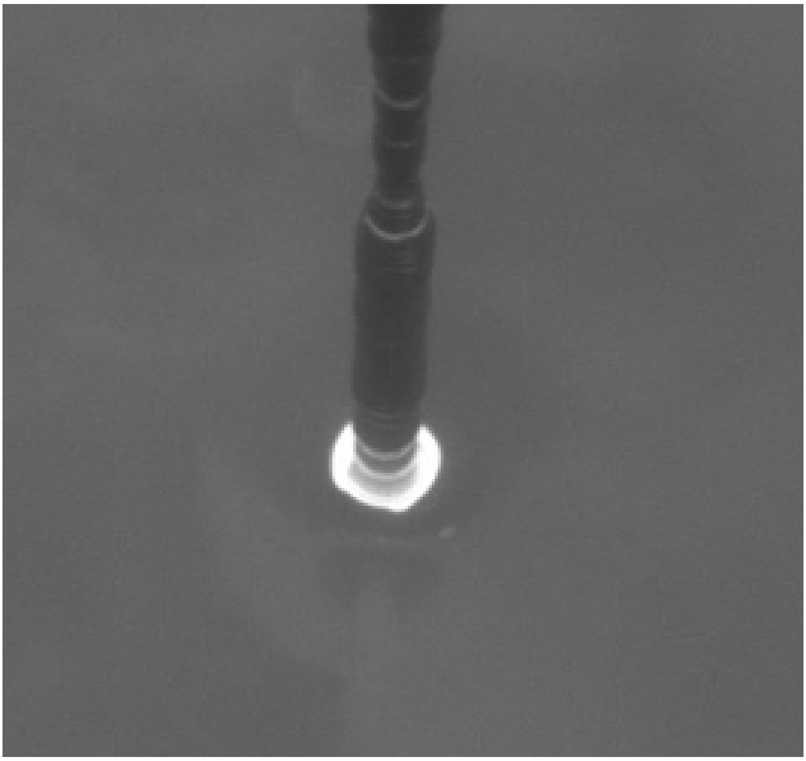


Figure 6.5: Image of a meniscus in a desired state one minute before starting the crown phase succesfully.

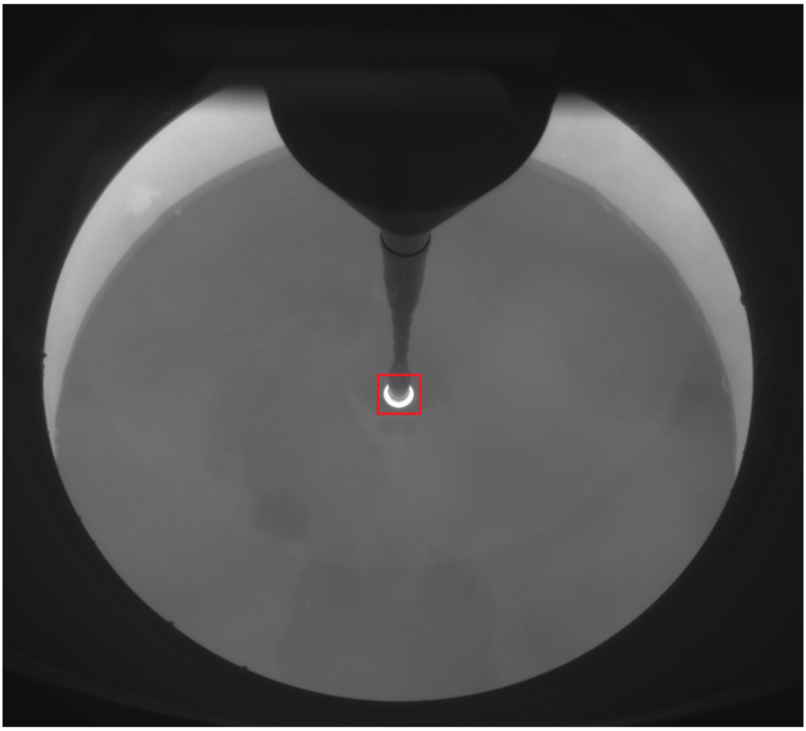


Figure 6.6: Image of an example *ROI* for a machine learning approach.

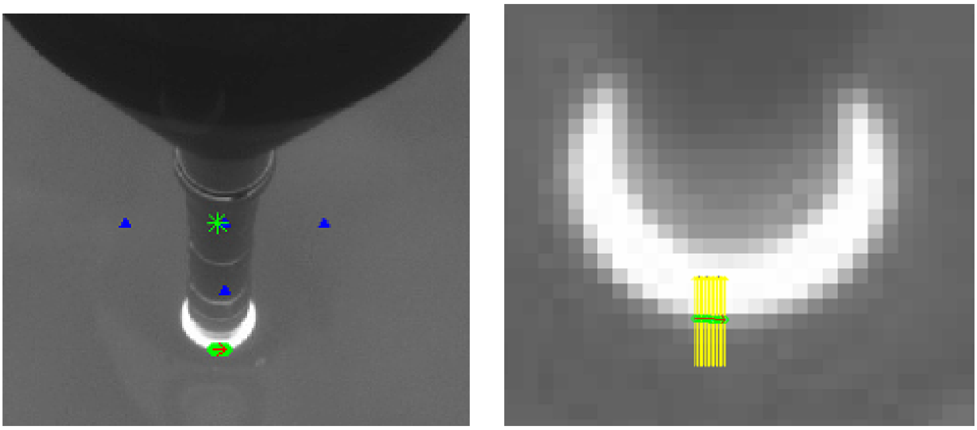
predict on is pretty small as illustrated in **Fig. 6.6**.

6.3 Results

6.3.1 Separate detection approach

Early in the development of the separate detection approach some problems occured. The *Scorpion Vision* tools used to detect the edges of the meniscus would sometimes fail or return badly placed edge positions. Most likely this is because the camera has a resolution of *1600x1200px* and the area of interest is within an approximate *60x60px* area. Within this *ROI* the scale is so small that the light intensity changes become pixelated and the edge detectors need pixel perfect position to find the edge and ensure consistent results between frames. The pulling mechanism holding the seed also sometimes orbit, which means it will not necessarily stay in the exact same position between images. This requires functionality to dynamically find the center of where the neck hits the meniscus with pixel perfect accuracy to use as reference for placing the edge detector as seen in **Figs. 6.7** and **6.8**.

Creating a robust detection algorithm for all the separate states shown in **Figs. 6.1 - 6.5** will require a very sophisticated algorithm for dynamically placing the center of the



(a) The LineFinder2 tool detecting the edge of the meniscus.

(b) A zoomed in view of the LineFinder2 tool detecting the edge of the meniscus.

Figure 6.7: The LineFinder2 edge detector finding the edge of the meniscus

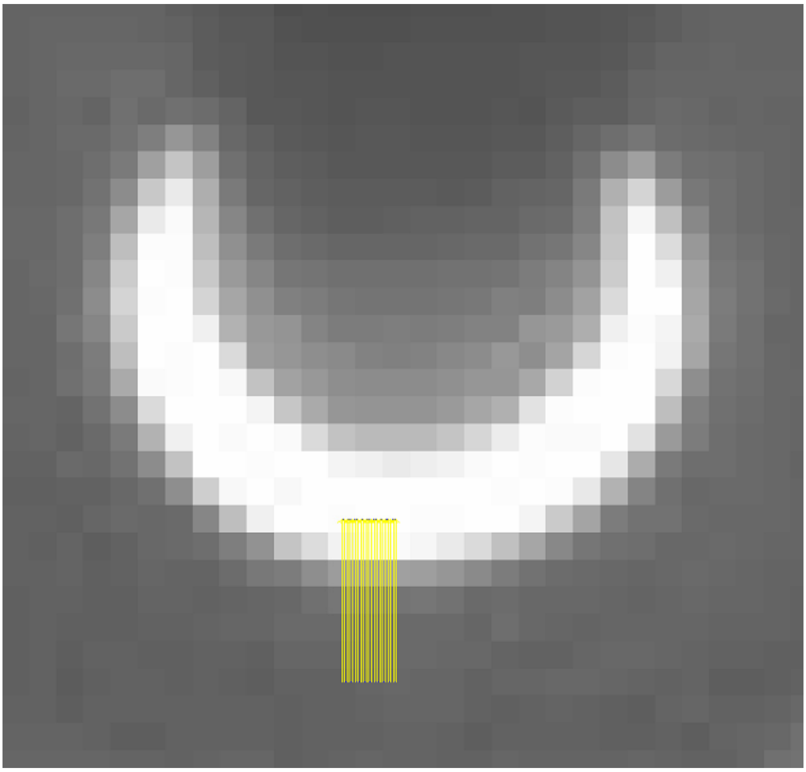


Figure 6.8: The LineFinder2 tool failing after being placed *1px* further down on the image from **Fig. 6.7b**.

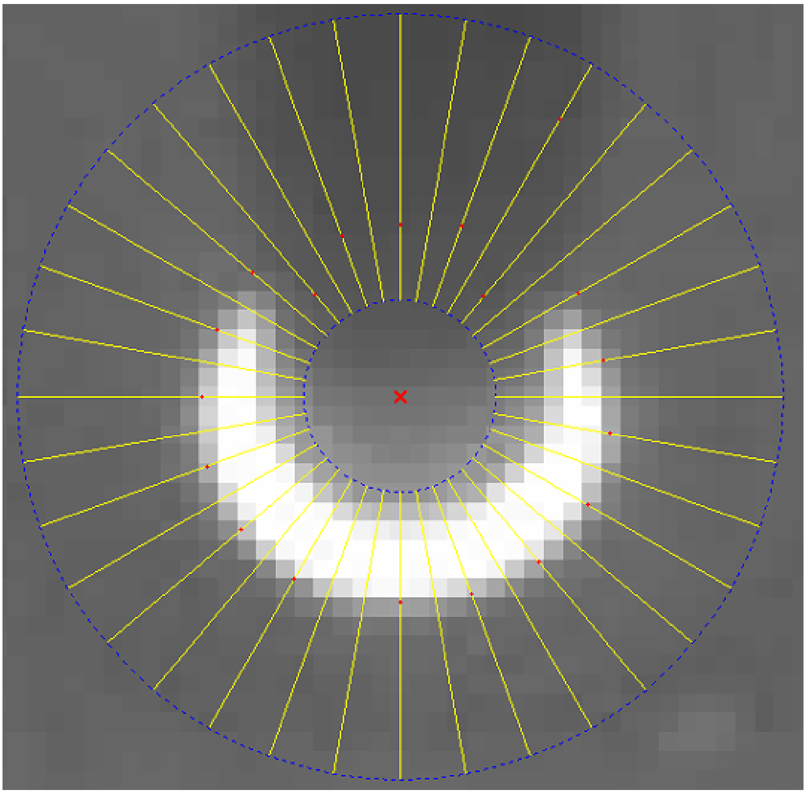


Figure 6.9: The RadialArcFinder tool failing to find enough edge points to fit a circle. The detected edge points are indicated by small red dots along the yellow tracelines.

image with pixel precision. An attempt was made by using the *RadialArcFinder* tool from *Scorpion Vision* as shown in **Fig. 6.9**.

The *RadialArcFinder* tool attempts to find edges in intensity along tracelines between the inner circle and the outer circle. These edges are used to fit a circle which should indicate the circle of the meniscus. If enough of the edges fail the tool can not return a circle with a satisfactory precision, and therefore fails. The *RadialArcFinder* could have been used to identify the center of the meniscus and used for placing the edge detector, but due to the low resolution the edge detector has trouble finding the exact edge with high enough precision.

Discussion

With a higher resolution camera the separate detection approach might be feasible. Currently it requires a level of precision the computer vision algorithms are not able to support within the current resolution. If a camera with a higher resolution was implemented it would still be a challenge, since detecting nodes at this step is similar and has the same level of difficulty as detecting nodes for the structure loss algorithm discussed in **Chap.**

5. And this is just one out of five states it aims to detect. This detection will require a lot of work to guarantee a high enough level of robustness to be implemented for use in the factory environment.

6.3.2 Machine learning approach

Another approach to this problem is to use machine learning instead of creating a detection algorithm for every single state that can occur during the neck phase. This approach requires an experienced operator or someone with training in detecting the states by visual inspection to manually classify a large data set of images from different states. These images then have to be cropped to the size of the *ROI*. Once the manual classification is done some research on how the machine learning model should be designed has to be done. This approach might work on the current resolution, but with a higher resolution the robustness of a machine learning based detection will definitely increase. If the image gets a higher resolution the machine learning model will be able to detect the small changes such as split nodes or change of size in the nodes more robustly. Combining this approach with a classification of the data resulting in a success of the full ingot growth might even let the algorithm itself find new variations in the image which gives the ideal initial conditions for the crown phase. Classifying different states in images like this is one of the main uses of machine learning in state-of-the-art computer vision applications. This should be an indicator that this method might be the way to go to achieve the best result of this detection.

This approach was never attempted due to time-to-deadline issues, so there is no data on this approach. This section is purely discussion based upon speculation with root in the theory presented in **Chap. 2**.

Chapter 7

Conclusions and recommendations for further work

7.1 Test of Cold Ingot Detection in the live environment

7.1.1 Conclusion

The detection of cold ingots is not able to give an early enough detection in its current state. There seems to be a disrepancy in the increase of *ingot_squareness* depending on how quickly the ingot becomes cold. If the change happens quickly *ingot_squareness* increases to above the alarm threshold and clearly indicates a cold ingot. If the ingot becomes cold over a longer period of time the *ingot_squareness* doesn't increase to a high enough value to activate the alarm. This might be remedied by tuning the threshold by look at graphs from *APIS* ensuring the max value of a non-cold ingot never exceeds the value chosen for the threshold. In the specific case where an ingot became cold without an alarm being activated, the temperature was adjusted while *ingot_squareness* had a value of circa 2.3. Using this as a baseline for a new threshold might ensure a satisfactory detection as long as it doesn't introduce any false positives.

7.1.2 Recommendation for further work

The future work for the test of cold ingot detection in the live environment requires access to the real time data in the factory, preferrably with possibilities of looking at every frame which is evaluated as well. Reducing the alarm threshold to around 2.35 might be enough to solve the problem. To ensure that this approach is good enough, tests have to be run and healthy ingots have to be analyzed to make sure their worst case *ingot_squareness* value never exceeds the suggested threshold. If this does not work then maybe another approach can be used in conjuction with the meniscus angle method which is currently used to guarantee a robust detection of cold ingots.

7.2 Detection of structure loss in the body phase

7.2.1 Conclusion

Structure loss is a difficult phenomenon to detect due to the meniscus changes being so miniscule that it is hard to discern the actually nodes from noise. If a method is not able to consistently detect the nodes without any false positives or positive negatives then the entire approach is not robust enough in a production setting. Out of the methods mentioned in **Chap. 5** the mean value approach and the machine learning approach seem like the only two methods to be able to solve this problem. The mean value approach is the easiest and least time consuming method to implement and it is also easy to test as long as the real time data from *APIS* is used for the analyzis. The machine learning method on the other hand is very unfinished and needs a lot more data, tweaking of the model and testing to be considered robust enough to work in a live environment. It also requires a better setup for creating the test data to ensure there are no discrepancies in the *ROI* placement between ingots.

7.2.2 Recommendation for further work

Since many of the attempts at detecting structure loss in the body phase are dependent on time series data, which is widely different in the test setup compared to the live factory environment, the person conducting the further work needs access to real time data and images from the factory. The recommended approaches are the mean value approach and the machine learning approach. The mean value approach should take a lot less time to implement and thus should be tried before starting work with the machine learning approach.

7.3 Automatic neck temperature detection concept

7.3.1 Conclusion

The detection of automatic neck temperature only focuses on a small portion of the image where there is a wide variety of problems and unwanted states that may arrive. This portion of the image is so small that picking up on the details with the current camera and classic comptuer vision techniques proves to be very difficult. If a camera with better resolution is installed, two main approaches for solving this detection problem are recommended. One of them is creating separate detection for each of the individual unwanted states, sending them all to a central logical controller which uses this information to decide if the temperature is too low, too high or at an appropriate level. These separate detections will vary in complexity but for example the detection of the nodes will be quite similar to the structure loss detection algorithm, only on a smaller scale. Creating separate detections for all the unwanted states in this phase might be very time consuming. Another approach is to design a machine learning algorithm and train it on a huge data set of manually classified images by experienced operators of these phenomena. It can either be a binary classification where the meniscus is either reported as a desired state or unwanted state, or a more complex classification where temperature adjustment recommendations can be returned depending on the resulting state. Both approaches will require a great deal of analysis and time to guarantee a satisfactory performance.

7.3.2 Recommendation for further work

For further work on the automatic neck temperature detection concept there has to be done research on which approach is most likely to yield the best response. Since both of the approaches are technically difficult and time consuming, their pros and cons should be carefully analyzed and discussed before picking a method. A camera with a higher resolution, either from applying an optical zoom or a camera with higher resolution, should be acquired to guarantee feasibility to the research done on this detection.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X., 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.
URL https://www.tensorflow.org/

Bones, J., 2012. Model-based control of the czochralski process.

Brakel, J., 2016. Smoothed z-score algorithm. http://stackoverflow.com/questions/22583391/peak-signal-detection-in-realtime-timeseries-data, [Online; accessed 20.05.2018].

Canny, J., 1986. A computational approach to edge detection. IEEE Transactions On Pattern Analysis And Machine Intelligence PAMI-8 (6).

Catalbas, M., Cegovnik, T., Sodnik, J., Gulten, A., 2017. Driver fatigue detection based on saccadic eye movements. 10th International Conference on Electrical and Electronics Engineering (ELECO), 913–917.

Chollet, F., et al., 2015. Keras. https://keras.io, [Online; accessed 12.05.2018].

Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep Learning. MIT Press, http://www.deeplearningbook.org.

Hunter, J. D., 2007. Matplotlib: A 2d graphics environment.

Jones, E., Oliphant, T., Peterson, P., et al., 2001–. SciPy: Open source scientific tools for Python. SciPy Home, [Online; accessed 02.05.2018].

Kakimoto, K., 2013. Development of crystal growth technique of silicon by the czochralski method.

Lanterne, A., Gaspar, G., Hu, Y., Øvrelid, E., Sabatino, M., 2016. Investigation of different cases of dislocation generation during industrial cz silicon pulling. Physica Status Solidi C (10-12), 827–832.

Lee, D., Park, J., Lee, K., Lee, M., 2005. Mpc based feedforward trajectory for pulling speed tracking control in the commercial czochralski crystallization process. International Journal of Control, Automation, and Systems 3 (2), 252–257.

Liew, L., 2016. Curve fitting. `https://algotrading101.com/blog/1543426/what-is-curve-fitting-overfitting-in-trading-optimization`, [Online; accessed 29.05.2018].

Mitchell, T., 1997. Machine Learning. McGraw-Hill.

NorSun, 2018. Norsun pulling hall. `https://norsuncorp.no/`, [Online; accessed 04.06.2018].

Prediktor, 2018. Prediktor webpage. About APIS link, [Online; Accessed: 25.04.2018].

Siemens, 2016. SIMATIC WinCC reference. WinCC Information Link, [Online; Accessed: 25.04.2018].

Sony, 2016. XCG-U100E brochure. Brochure Link, [Online; Accessed: 28.04.2018].

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salkhutdinov, R., 2014. Dropout: A simple way to prevent neural networks from overfitting (15), 1929–1958.

Szeliski, R., 2010. Computer Vision: Algorithms and Applications. Springer.

Tao, Q., I, V., Yuesheng, X., 2007. Wavelet Analysis and Applications. BIRKHÄUSER.

Valens, C., 1999. A Really Friendly Guide to Wavelets. BIRKHÄUSER.

Appendix

Meniscus angle method from previous project in 2017

```
# OLD METHOD USED IN PREVIOUS WORK
# MENISCUS ANGLE METHOD
##Calculate standard deviation of angle on 3 lines along the meniscus

# Horizon = 15 frames
N = 15

#Fetch values from edge detectors, and fetch horizon iterator
itr = GetIntValue(’IngotShape.Counter’)
meniscus1_angle = GetValue(’MeniscusLineFinder1.Line[1]_Angle’)
meniscus2_angle = GetValue(’MeniscusLineFinder2.Line[1]_Angle’)
meniscus3_angle = GetValue(’MeniscusLineFinder3.Line[1]_Angle’)

angleList1 = GetResultValue(’IngotShape.AngleList1’)
angleList2 = GetResultValue(’IngotShape.AngleList2’)
angleList3 = GetResultValue(’IngotShape.AngleList3’)

#Gather first 15 sets before starting calculation
if itr < 15:
  angleList1.append(meniscus1_angle)
  angleList2.append(meniscus2_angle)
  angleList3.append(meniscus3_angle)

  SetResultValue(’IngotShape.AngleList1’, angleList1)
  SetResultValue(’IngotShape.AngleList2’, angleList2)
  SetResultValue(’IngotShape.AngleList3’, angleList3)
  itr += 1
  SetIntValue(’IngotShape.Counter’, itr)
#After 15 points check std.deviation
else:

  #Get means
  mean1 = np.sum(angleList1)/N
  mean2 = np.sum(angleList2)/N
  mean3 = np.sum(angleList3)/N
  s1 = 0
  s2 = 0
  s3 = 0
  for i in range(0, N):
    s1 += np.power(np.absolute(angleList1[i]-mean1), 2)
    s2 += np.power(np.absolute(angleList2[i]-mean2), 2)
    s3 += np.power(np.absolute(angleList3[i]-mean3), 2)
  # Calc std.dev of each vector
```

```
      sdev1 = np.sqrt(s1/N)
      sdev2 = np.sqrt(s2/N)
      sdev3 = np.sqrt(s3/N)
      mean = (sdev1+sdev2+sdev3)/3
      SetValue(’TOOLBOX.MeanSDev’, mean)

      # Remove oldest frame from history and append new for next iteration
      del angleList1[0]
      del angleList2[0]
      del angleList3[0]
      angleList1.append(meniscus1_angle)
      angleList2.append(meniscus2_angle)
      angleList3.append(meniscus3_angle)
      SetResultValue(’IngotShape.AngleList1’, angleList1)
      SetResultValue(’IngotShape.AngleList2’, angleList2)
      SetResultValue(’IngotShape.AngleList3’, angleList3)
```

New meniscus angle method

```
#IngotShape detector for cold ingots by Meniscus Angle

#Horizon, #frames
N = 30

#Fetch values from edge detectors, and fetch horizon iterator
WriteCounter = GetIntValue(’StandardDeviation.WriteCounter’)
itr = GetIntValue(’StandardDeviation.Counter’)
meniscus1_angle = GetValue(’MeniscusLineFinder1.Line[1]_Angle’)
meniscus2_angle = GetValue(’MeniscusLineFinder2.Line[1]_Angle’)
meniscus3_angle = GetValue(’MeniscusLineFinder3.Line[1]_Angle’)

angleList1 = GetResultValue(’StandardDeviation.AngleList1’)
angleList2 = GetResultValue(’StandardDeviation.AngleList2’)
angleList3 = GetResultValue(’StandardDeviation.AngleList3’)

#Used to export data for testing in external program.
#SDArray = GetResultValue(’StandardDeviation.SDevList1’)

#Fetch status of MeniscusLineFinders, 1 = success, 3 = fail
status1 = GetIntValue(’MeniscusLineFinder1.Status’)
status2 = GetIntValue(’MeniscusLineFinder2.Status’)
status3 = GetIntValue(’MeniscusLineFinder3.Status’)

#If N is reduced during a run the program will adjust
if itr > N:
  del angleList1[0]
  del angleList2[0]
  del angleList3[0]
  itr -= 1
  SetIntValue(’StandardDeviation.Counter’, itr)

#Gather first N sets before starting calculation
#Don’t append unless all three tools are able to fetch values: statusX ==
                                           1
```

```
#This ensures three vectors with same length as the basis
if itr < N and status1 == 1 and status2 == 1 and status3 == 1:
  angleList1.append(meniscus1_angle)
  angleList2.append(meniscus2_angle)
  angleList3.append(meniscus3_angle)
  SetResultValue('StandardDeviation.AngleList1', angleList1)
  SetResultValue('StandardDeviation.AngleList2', angleList2)
  SetResultValue('StandardDeviation.AngleList3', angleList3)
  itr += 1
  SetIntValue('StandardDeviation.Counter', itr)

#After N points, check std.dev
elif itr == N:
  sdev1 = np.std(angleList1)
  sdev2 = np.std(angleList2)
  sdev3 = np.std(angleList3)
  mean = (sdev1+sdev2+sdev3)/3
  SetValue('TOOLBOX.MeanSDev', mean)
  # These values can be written to OPC to look at cold ingot detection-
  # problems.
  SetValue('TOOLBOX.SDev1', sdev1)
  SetValue('TOOLBOX.SDev2', sdev2)
  SetValue('TOOLBOX.SDev3', sdev3)


  ## Used to extract standard deviation results to external scripts-
  ## for analysis.
  #if len(SDArray) >= 30:
  #  del SDArray[0]
  #SDArray.append(sdev1)
  #SetResultValue('StandardDeviation.SDevList1', SDArray)

  #Remove oldest frame from history and append new for next itr
  #Only allow update of value if the tool doesn't fail.
  if status1 == 1:
     del angleList1[0]
     angleList1.append(meniscus1_angle)
     SetResultValue('StandardDeviation.AngleList1', angleList1)
  if status2 == 1:
     del angleList2[0]
     angleList2.append(meniscus2_angle)
     SetResultValue('StandardDeviation.AngleList2', angleList2)
  if status3 == 1:
     del angleList3[0]
     angleList3.append(meniscus3_angle)
     SetResultValue('StandardDeviation.AngleList3', angleList3)


#Try-Except block to catch failure in OPC link.
try:
  # Only count towards write when list is populated and mean calculated
  # Upper limit on mean is safety in case some error making linefinders
                                        fail
  if itr == N and mean != 0 and mean < 10:
    # 4 updates to OPC between full reset of data in populated list.
    # WriteCounter accept value must be tuned depending on N and desired
                                          write cycle
    if WriteCounter >= N/2:
      WriteCounter = 0
      #Convert to int with 3 decimal precision
      mean_out_int = int(mean*1000)
      print 'SEND SDEV TO OPC'
      opc.WriteItem('Ingot_Squareness', mean_out_int)
    # Iterate up writecounter
    else:
      WriteCounter += 1
    #Always update WriteCounter
    SetIntValue('StandardDeviation.WriteCounter', WriteCounter)
  #if itr != N or mean has unreasonable value, don not write to OPC.
  else:
    pass
except:
  #If error in OPC link - should something happen? Reset all variables and
                                        restart??
  pass
```

Derivative script for structure loss detection test

```
import matplotlib.pyplot as plt
import numpy as np
import random

# Contains sdev values from cold ingot detection.
y = np.array([])




ymin = np.amin(y)
ymax = np.amax(y)
## Set up plots
fig, ax = plt.subplots(3, 1)# figsize=(8,6))

# Initialize empty signal array with correct length
siglist = np.zeros(len(y))

# Find positive zero-crossings of the slope.
def crossings_nonzero_pos(data):
    pos = data > 0
    # Binary AND between the entire set of data > 0
    return (pos[:-1] & ˜pos[1:]).nonzero()[0]

# Find negative zero-crossings of the slope.
def crossings_nonzero_neg(data):
    pos = data < 0
    return (pos[:-1] & ˜pos[1:]).nonzero()[0]

# Set signal high on correct index on detected cross
def signal_crossing_pos(data, siglist, slope):
    for i in range(0, len(data)):
        siglist[data[i]] = 1
    return siglist

# Derivative
slope = np.gradient(y)
# Find all zero crossings where sign of slope is changed
zero_crossings = np.where(np.diff(np.sign(slope)))[0]

# Pick all signs going from positive to negative from set of all zero
                                      crossing
zero_cross_pos = crossings_nonzero_pos(slope)
# Pick all signs going from neg to pos of all zero crossing
zero_cross_neg = crossings_nonzero_neg(slope)

# For every peak there has to be a dip so we can do:
for i in range(1, len(zero_crossings), 2):

    if i < len(zero_crossings)-1 and i != 0:
        temp_a = y[zero_crossings[i]]
        temp_b = y[zero_crossings[i+1]]
        print("First frame: ", zero_crossings[i], "Second frame: ",
                                              zero_crossings[i+1])
        print("Diff: ", np.abs(temp_a - temp_b))


# Translate result to a plottable signal array
sig_result = signal_crossing_pos(zero_cross_pos, siglist, slope)

# Plotting
ax[0].plot(y, color='blue')
ax[0].set_title("Standard deviation")
ax[0].set_ylabel("Standard Deviation", color='blue')
ax[0].set_xlim([0, len(y)])
ax[0].set_ylim([ymin-0.2, ymax+0.2])
ax[0].grid()
ax[1].plot(slope, color='green')
ax[1].set_title("Slope of standard deviation")
ax[1].set_ylabel("Slope", color='green')
ax[1].set_xlim([0, len(y)])
ax[1].set_ylim([-0.25, 0.25])
ax[1].grid()
ax[2].plot(sig_result, color = 'red')
ax[2].set_title("Signal")
ax[2].set_ylabel("Signal peaks", color='red')
ax[2].set_xlim([0, len(y)])
ax[2].set_ylim([0, 1.5])
ax[2].grid()
plt.show()
```

Fourier transform script

```
# Fourier transform
import scipy
from scipy import signal
import matplotlib.pyplot as plt
import numpy as np

# Contains sdev values from cold ingot detection.
y = np.array([])

# Size of data
N = len(y)-1

# Arbitrary value to spread the resulting plot
T = 1/30
# Size of window
batch = 30

# Generate an X axis
x = np.linspace(0.0, N*T, N)
# Create window to transform within
wind = signal.get_window(’blackman’, batch)

#Short time fourier transform
f, t, Zxx = signal.stft(y, window = wind, nperseg=batch)

#Plot setup
plt.title(’STFT Magnitude’)
plt.ylabel(’Frequency [HZ]’)
plt.xlabel(’Time [sec]’)

plt.pcolormesh(t, f, np.abs(Zxx), vmin=0, vmax=2)
plt.show()
```

Wavelet transform script

```
# Wavelet transform
import scipy
from scipy import signal
import matplotlib.pyplot as plt
import numpy as np

# Contains sdev values from cold ingot detection.
y = np.array([])

# Wavelet transform
peakidx = signal.find_peaks_cwt(y, np.arange(1,5), noise_perc = 0.1)

plt.subplot(211)
plt.plot(y)
plt.plot(peakidx, y[peakidx], ’x’)
plt.show()
```

Smoothed Z-score algorithm

```
# Implementation of algorithm from http://stackoverflow.com/a/22640362
                                                              /6029703
import numpy as np
import pylab

def thresholding_algo(y, lag, threshold, influence):
    signals = np.zeros(len(y))
    filteredY = np.array(y)
    avgFilter = [0]*len(y)
    stdFilter = [0]*len(y)
    avgFilter[lag - 1] = np.mean(y[0:lag])
    stdFilter[lag - 1] = np.std(y[0:lag])
    for i in range(lag, len(y) - 1):
        if abs(y[i] - avgFilter[i-1]) > threshold * stdFilter [i-1]:
            if y[i] > avgFilter[i-1]:
                signals[i] = 1
            else:
                signals[i] = -1

            filteredY[i] = influence * y[i] + (1 - influence) *
                                                 filteredY[i-1]
            avgFilter[i] = np.mean(filteredY[(i-lag):i])
            stdFilter[i] = np.std(filteredY[(i-lag):i])
        else:
            signals[i] = 0
            filteredY[i] = y[i]
            avgFilter[i] = np.mean(filteredY[(i-lag):i])
            stdFilter[i] = np.std(filteredY[(i-lag):i])

    return dict(signals = np.asarray(signals),
                avgFilter = np.asarray(avgFilter),
                stdFilter = np.asarray(stdFilter))


# Data
y = np.array([data..])

# Settings: lag = 30, threshold = 5, influence = 0
lag = 2
threshold = 1
influence = 0.7

# Run algo with settings from above
result = thresholding_algo(y, lag=lag, threshold=threshold, influence=
                                                              influence)

f, ax = pylab.subplots(3, sharex = True)

ax[0].plot(np.arange(1, len(y)+1), y)
ax[0].set_title('Data set')
ax[1].plot(np.arange(1, len(y)+1), y)
ax[1].plot(np.arange(1, len(y)+1),
           result["avgFilter"], color="cyan", lw=2)
ax[1].plot(np.arange(1, len(y)+1),
           result["avgFilter"] + threshold * result["stdFilter"],
                                            color="green", lw=2)
ax[1].plot(np.arange(1, len(y)+1),
           result["avgFilter"] - threshold * result["stdFilter"],
                                                color="green", lw=2)
ax[1].set_title('Data set with smoothed Z-score')
ax[2].step(np.arange(1, len(y)+1), result["signals"], color="red", lw=
                                     2)

pylab.show()
```

Machine learning training script

```
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.preprocessing.image import ImageDataGenerator
from keras.layers import Activation, Dropout, Dense, Flatten
from keras.layers import Conv2D, MaxPooling2D, BatchNormalization
from keras.utils import plot_model
from keras import optimizers
import matplotlib.pyplot as plt

#dims
img_width = 64
img_height = 64
#Training consts
batch_size = 32
epochs = 50
train_samples = 140
valid_samples = 32

#Black & white
input_shape = (img_width, img_height, 3)

#Directories of image sets
train_dir = 'data/train'
valid_dir = 'data/validation'

#Feedforward model, add layers in sequential order
model = Sequential()

#Layer 0, 32 inputs, 0.5 dropout
model.add(Dense(32, input_shape=input_shape))
model.add(Activation('relu'))
model.add(Dropout(0.5))

#Layer 1, 16 inputs, 0.5 dropout
model.add(Dense(16, activation='relu'))
model.add(Dropout(0.5))
model.add(Flatten())

#Layer 2, fully connected sigmoid 0-1 result
model.add(Dense(1, activation='sigmoid'))

#Compiler
model.compile(optimizer='nadam', loss='binary_crossentropy', metrics=[
                                     'accuracy'])
```

```
#Generate training data from images
if True:
    train_datagen = ImageDataGenerator(
        rescale = 1./255,
        )

    test_datagen = ImageDataGenerator(
        rescale = 1./255,
        )

    train_gen = train_datagen.flow_from_directory(
        train_dir,
        target_size = (img_width, img_height),
        batch_size = batch_size,
        class_mode = 'binary'
        )

    valid_gen = test_datagen.flow_from_directory(
        valid_dir,
        target_size = (img_width, img_height),
        batch_size = batch_size,
        class_mode = 'binary'
        )

#Train the model on the new data for a few epochs
history = model.fit_generator(
        train_gen,
        steps_per_epoch = train_samples // batch_size,
        epochs = epochs,
        verbose = 1,
        validation_data = valid_gen,
        validation_steps = valid_samples // batch_size
        )

#Save training model to file
model.save_weights('model_weights.h5')
model.save('model.h5')

#Evaluate score of model
score = model.evaluate_generator(valid_gen)
print("Score: ", score)
print("%s: %.2f%%" % (model.metrics_names[1], score[1]*100))

#Plot training history
plt.figure(1)

#Summarize history for accuracy
plt.subplot(211)
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')

#Summarize history for loss
```

```
plt.subplot(212)
plt.plot(history.history[’loss’])
plt.plot(history.history[’val_loss’])
plt.title(’model loss’)
plt.ylabel(’loss’)
plt.xlabel(’epoch’)
plt.legend([’train’, ’validation’], loc=’upper left’)
plt.show()
```

Machine learning prediction from model script

```
from keras.models import load_model
from keras.preprocessing import image
from keras.preprocessing.image import ImageDataGenerator
import numpy as np
import matplotlib.pyplot as plt

#Fetch best scoring model
model = load_model(’C:\\Users\\endreag\\AnacondaProjects\\rename_files
                                      \\model_96.h5’)

#Fetch directory of images to predict
pred_dir = ’data/prediction’
#Constants
img_width = 64
img_height = 64
batch_size = 1
pred_samples = 79

#Generate data from images in directory
pred_datagen = ImageDataGenerator(
    rescale = 1./255,
    )
#Generate prediction from the model on the data
pred_gen = pred_datagen.flow_from_directory(
    pred_dir,
    target_size = (img_width, img_height),
    batch_size = batch_size,
    class_mode = ’binary’,
    shuffle = False
    )


#Get the class indices from generator
ground_truth = pred_gen.classes

#Get the filenames from the generator
fnames = pred_gen.filenames

#Get the label to class mapping from the generator
label2index = pred_gen.class_indices

#Getting the mapping from class index to class label
idx2label = dict((v,k) for k,v in label2index.items())
```

```
    #Find probabilities
    pred_prob = model.predict_generator(pred_gen, verbose=1, workers = 1)

    #Fetch predicted classification
    pred_class = np.reshape(np.where(pred_prob > 0.5, 1, 0), (-1))
    print("Pred class: ", pred_class)

    #Find all wrongly classified images.
    errors = np.where(pred_class != ground_truth)[0]
    print("Num of errors: ", len(errors))
    print(errors)
    #Find all correctly classified images:
    correct = np.where(pred_class == ground_truth)[0]
    print("Num correct: ", len(correct))

    # Shows the image indicated as an error with predicted class and
                                     actual class as label
    for i in range(len(errors)):
        error_class = pred_class[errors[i]]
        #print("Pred class: ", pred_class)
        pred_label = idx2label[error_class]
        #print("Pred label: ", pred_label)
        print("Filename: ", fnames[errors[i]])

        title = 'Original label: {}, Prediction: {}'.format(
            fnames[errors[i]].split('/')[0],
            pred_label
            )

        original = image.load_img('{}/{}'.format(pred_dir,fnames[errors[i]
                                         ]))
        plt.figure(figsize=[7,7])
        plt.axis('off')
        plt.title(title)
        plt.imshow(original)
        plt.show()
```

Utility tool for renaming image files as a single iterated number

```
import os
import sys

def renameAllFilesInFolder(folderpath):
    #Name of the first image in the set = i
    i = 1
    for filename in os.listdir(folderpath):
        oldFile = os.path.join(folderpath, filename)
        if os.path.exists(oldFile):
            #Path magic for filename and file extension concatenation
            newFile = os.path.join(folderpath, str(i)+".png")
            os.rename(oldFile, newFile)
```

```python
        else:
            print("Couldn't find files in path.")
            return
        #Iterate i to avoid overwriting.
        i += 1

def main():
    #Avoid running if path not included in call
    if(len(sys.argv) <= 1 or len(sys.argv) > 2):
        exit
    else:
        pathStr = str(sys.argv)
        print("Looking to rename files in path: ", sys.argv[1])
        renameAllFilesInFolder(str(sys.argv[1]))
main()
```