CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Quality Monitoring in AM Metal Through Efficient Streaming Machine Learning

A low-latency in-situ monitoring pipeline to locate and study suspicious areas

Master's thesis in Computer science and engineering

Hampus Palm
Venkata Sai Dinesh Uddagiri

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY

Gothenburg, Sweden 2024

Master's thesis 2024

Quality Monitoring in AM Metal Through Efficient Streaming Machine Learning

A low-latency in-situ monitoring pipeline to locate and study suspicious areas

Hampus Palm

Venkata Sai Dinesh Uddagiri

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Quality Monitoring in AM Metal Through Efficient Streaming Machine Learning
A low-latency in-situ monitoring pipeline to locate and study suspicious areas
Hampus Palm
Venkata Sai Dinesh Uddagiri

Supervisor: Marina Papatriantafilou, Department of Computer Science and Engineering
Co-Supervisor: Erik Sievers, Department of Computer Science and Engineering
Examiner: Vincenzo Gulisano, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Quality Monitoring in AM Metal Through Efficient Streaming Machine Learning
A low-latency in-situ monitoring pipeline to locate and study suspicious areas
Hampus Palm
Venkata Sai Dinesh Uddagiri
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Additive Manufacturing (AM), popularly known as 3D printing, has become a transformative technology with wide-ranging applications in a number of industries such as medical, aerospace, and automotive. The increasing adoption of AM in these critical sectors necessitates a focus on ensuring the quality of printed components. An emerging method to estimate the quality of an object is to process data coming out of the printer during printing. This data can then help detect defects within the object. Most of the current research focuses on the ability and accuracy of finding defects but fails to take into account the crucial time aspect associated with processing the data coming from the printer. This thesis proposes a method of quality monitoring in AM through the integration of machine learning and low-latency data-stream processing. Monitoring live sensor data from inside the printer (in-situ) has been advocated to minimize time for defect detection, yet its successful implementation relies on establishing a quantitative relationship between sensing data and the defects in an object. This proves challenging given the multitude of variables involved with this specific type of AM method.

This thesis utilizes Optical Tomography (OT) images captured as the object is being printed. The OT images are used together with spatial algorithms and machine learning models to identify defects in objects. The entire process is implemented in a streaming fashion, enabling low-latency monitoring and assessment. Importantly, the proposed approach supports not only the detection and position of defects but also the size of the defects, addressing both aspects that are critical for assessing the quality of AM components. The proposed methodology contributes to the advancement of in-situ quality control in AM, addressing the critical need for timely defect detection and ensuring the production of high-quality components. The output includes the location of defects, and the geometric attributes of porous areas across the current layer, considering the adjacent layers in the object during the manufacturing process.

The proposed method for finding defects has a sub-quadratic processing overhead in the size of the input. In this work it was tested with data of multiple objects and images for 250 layers; with a recall of up to 95%, its running time for finding overlapping clusters in two adjacent layers is a few milliseconds, hence allowing significant margin for closed-loop control.

Keywords: Outlier Detection, Machine learning, Optical Tomography, In-situ Sensing

Acknowledgements

We would like to thank Marina Papatriantafilou and Vincenzo Gulisano for their investment, support, and input during the course of this thesis. We would also like to thank our co-supervisor Erik Sievers for his continuous support and feedback throughout this entire work. Additionally, we extend our thanks to Binsha Nazar, Sangeet John, Carl-Magnus Wall, and Måns Josefsson for your feedback during the presentations of this work. We would like to acknowledge the Powder metallurgy and additive manufacturing (PMAM) research group of the Department of Industrial and Material Science who provided data for the project, and the STRATIFIER WISE/WASP seed project that supported part of the supervision activities. We finally want to thank our family and loved ones for supporting us through the entire process.

Hampus Palm, Venkata Sai Dinesh, Gothenburg, 2024-09-08

Contents

List of Figures

List of Tables

Glossary

- **AM** - Additive Manufacturing refers to the process of creating a three-dimensional object by adding materials through various techniques, as opposed to removing material.
- **AABB** - Axis-Aligned Bounding Box, a bounding box where the edges are the minimum and maximum of each dimension.
- **ML** - Machine Learning
- **L-PBF** - Laser Powder Bed Fusion, a type of additive manufacturing utilizing a laser to melt a metal powder into a solid object.
- **DBSCAN** - Density-based Spatial Clustering of Applications with Noise, a clustering algorithm using the density of data points to build clusters.
- **OT** - Optical Tomography, a non-invasive imaging technique that uses light to capture detailed, cross-sectional images of the internal structure of an object.
- **Stream Processing**, the real-time processing of continuous data streams to enable immediate analysis and response.
- **DAG** - Directed Acyclic Graph, graphs with directed edges and no cycles, meaning they do not contain loops.

1

Introduction

1.1 Additive Manufacturing

Additive Manufacturing (AM), colloquially known as 3D printing, has revolutionized modern manufacturing by offering unprecedented design flexibility and customization. Among the various AM techniques, *Laser Powder Bed Fusion* (L-PBF) stands out for its efficacy in producing intricate metal components with precise mechanical properties. As the demand for high-quality, complex, and customized metal parts continues to grow, optimizing the L-PBF process becomes paramount.

The L-PBF process involves the layer-by-layer deposition of metal powder, selectively melted by a focused laser beam. This approach allows for the creation of intricate designs and novel geometric configurations, making L-PBF particularly attractive for the precise manufacturing of complex structures [1]. However, achieving consistent quality in L-PBF presents challenges that traditional post-process control methods struggle to address. In L-PBF manufacturing, defects such as balling, porosity, spatters, geometric defects, and surface roughness, may arise. Of particular concern is porosity, a defect marked by voids or gaps that can significantly impact mechanical properties and spatters, particles ejected from the melt pool[2][3]. Both these defects occur in regions of high thermal energy.

Current quality control practices rely heavily on post-process defect detection, in which an expert analyzes the quality of the printed object using a set of data attained by sensors during the print and using techniques such as Visual Inspection, Ultrasonic Testing, or Radiographic Testing [4]. Post-process analysis is inefficient and costly as it may lead to the identification of defects only after the completion of the print, resulting in wasted resources and time.

1.2 In-situ Monitoring

In contrast to post-processing techniques, *in-situ monitoring* aims to enable control of the quality of the object as it is being constructed. To achieve this in-situ monitoring retrieves and processes data from the printer as it is printing. It takes advantage of the layer-wise production that is inherent for AM to detect flaws in each layer. This provides a more informative view of the object as it contains information about the inside of it. The interest in the subject has risen greatly in the last couple of

years as the possibility of using a closed-looped system, meaning having real-time feedback to the printer as it prints, has emerged [5].

To overcome the challenges of post-print analyzing and the need to re-print, proactive, data-driven monitoring is essential, as highlighted by the correlation between defects and printing parameters [5]–[7]. To make use of this correlation data analytics needs to be used to summarize and capture patterns in the data. This should be done with high accuracy and low latency to compete with the current, more manual, analysis. This is challenging because the data is often vast, complicating timeliness, and often quite heterogeneous, requiring small margins of error. The sensors in the printer, including acoustic sensors, high-resolution optical cameras, high-speed cameras, infrared thermography, and IR cameras, enable continuous data acquisition during the manufacturing process. Realizing real-time error identification necessitates online analysis of the data produced by these sensors. One of the most informative types of sensor data is infrared thermography. This is because it gives a complete 2D image of the thermal radiance in each layer [8].

1.3 Stream Processing

To effectively monitor live data that is produced by the AM machine there is a need to process it continuously. This paradigm of processing is called *stream processing*. Stream processing means processing live data in a continuous fashion to enable support for immediate decision-making and aggregating data for long-term analysis. In the context of this work, it means to process the data as it comes out of the sensors from inside the printer [9].

The use of stream processing technology along with incremental algorithms emerges as a crucial aspect of this thesis, as time to process the layer data is a deciding factor of the implementation's usefulness. Stream processing also includes tools to keep state using time or event-based windows allowing for comparisons between different data segments, e.g. data from 2 layers in an L-PBF print. More details about stream processing and the stream processing framework used in this thesis are given in Section 2.3.

1.4 Problem Statement

As mentioned above, there are several types of sensors in AM such as melt pool monitoring, laser power monitoring, etc., that provide complementary information. However, this thesis uses optical tomography images from a thermal camera inside an L-PBF printer. These images provide a 2D representation of each layer and show the heat signature, which is particularly useful since some defects in AM correlate with temperature [7]. So given a stream of images from a thermal camera situated inside an L-PBF printer the problem is locating suspicious areas, i.e. areas that possibly represent defects, in the print using the thermal values retrieved. This has to be done in a timely fashion as there is an inherent deadline as the printer keeps working on future layers and fixing errors becomes more difficult as time progresses.

An example of the image outputted by the thermal camera inside the printer is shown in Figure 1.1.

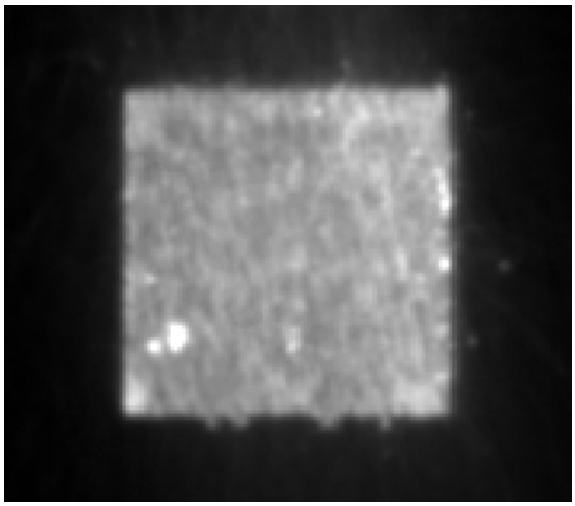


Figure 1.1: A thermal image of an object printed in an L-PBF printer where the intensity of the image indicates the thermal output of that area.

The *overarching objective* of the system is to output the suspicious areas of each layer and the geometry of suspicious areas across layers (in 3D) with low latency and high throughput analysis. Given the dynamic nature of the AM process, precise defect identification and localization are crucial for effective quality monitoring and ensuring the integrity of the final printed objects.

1.4.1 Problem Modeling

In L-PBF the laser transitions the powder bed to melt the powder into a solid. As the laser melts the powder, different parts can get a different thermal signature due to their geometry or input parameters. If the thermal signature of an area deviates from the rest of the object that area can be considered suspicious for defects. This means that the area has a higher probability of having a defect.

Without sensors and data from within the printer, defects can only be found after printing by having an expert examine the objects either by sawing them apart and looking for defects or using expensive X-ray equipment to give an inside look at them. This process is time-consuming and a waste of resources if it is found that the object is defective and not usable. Therefore it would be preferable to have a larger part of the process automated and be able to detect defects during printing.

In this thesis, we create a pipeline that uses in-situ monitor data in the form of thermal images to, during printing, find deviating thermal points, i.e. thermal outliers, and cluster them to quantify suspicious areas. Furthermore, we compare these clusters across layers to combine them into inter-layer clusters. Since what is most interesting about these clusters is the area/volume they span we also do a geometric summary of them using axis-aligned bounding boxes (AABB). This gives an expert

the possibility to use simple summaries of suspicious areas to quantify the quality of the print, automating and simplifying the decision to discard prints/objects.

In L-PBF there is also an inherent time aspect in analyzing the data as the printer keeps on working on the next layer and the longer time it takes to realize the quality of an object the more material and time is wasted. Because of this, we implemented this method of clustering in a streaming pipeline to allow for processing the images as they are created and to do so with low latency and high throughput.

1.4.2 Problem Formulation

Given a set of thermal images $\{S_1, S_2, ..., S_n\}$ from a camera situated inside the printer and where S_i represents the pixels of layer i, we want to identify the outlier pixels in each layer and cluster them into a set of 2D clusters. Continuing we want to combine the clusters of adjacent layers i and j into a set of 3D clusters C such that the size of each cluster $|C_{ij}| > minPts$, where $minPts$ is a predetermined minimum number of points required to call C_{ij} a cluster. And for each 2D sub-cluster c_i and c_j in C_{ij} there is at least one pair of points p_i and p_j where $|p_i - p_j| < \epsilon$, where ϵ is a predetermined distance to consider points to belong to the same cluster. These 3D clusters represent the suspicious or defective areas across layers of the object. This is to be done in a low-latency fashion and be able to classify potential defects with high accuracy.

1.5 Challenges

The main challenge of the thesis is the identification and quantification of suspected defects, that is, classifying a region as a defect and assessing the size of that region. This requires knowledge of the physical phenomenons that cause defects. Another challenge is the development of an incremental algorithm that utilizes stream processing to continuously compare images from different layers and merge suspected defects of one layer with suspected defects of an adjacent layer to build a view of defects across the layers throughout the object. A further challenge is the timeliness that the processing will need due to the inherent deadline as the printer keeps producing a specimen. This is challenging since the rates of data can be high with several images being taken per layer.

2

Preliminaries

In this section, the background information of this thesis is explained. First, we give an introduction to AM and specifically the type of AM from which the data in this thesis is gathered. After that, stream processing and its basics are covered to give a better understanding of the methodology of this work. We then introduce some clustering and data summary concepts including axis-aligned bounding boxes used to summarize the shape of suspicious areas, and lastly, we explain classification using machine learning.

2.1 Overview of Additive Manufacturing (AM)

Additive manufacturing (AM) is a method of manufacturing that consists of adding materials together to create an object rather than, as done traditionally, removing material to get the desired result, as is the case using a lathe or mill. One commonly known form of AM is 3D printing by melting a thin wire of plastic and layer by layer building an object described by a 3D modeling file.

As advancements continue in AM, a wide array of materials have been developed for 3D printing. These materials encompass metals, polymers, composites, ceramics, wood, powders, glass, building materials, biodegradable materials, smart materials, and more [10]. Each material is compatible with specific 3D printing methods tailored for particular applications.

2.1.1 Introduction to L-PBF

There are many different types of AM methods but in this thesis, our research is specific to L-PBF AM which is a powder-based AM process. It works by having a bed of powder that is usually some form of metal alloy and selectively melting and fusing the metal powder material layer by layer, guided by a 3D digital model. This process enables the creation of intricate metal parts with high precision and customizable geometries.

In the L-PBF process, a powder bed is initially created by depositing powder across a substrate or platform using a recoater. A computer-controlled electron or laser beam energy source then supplies thermal energy to the surface of the powder bed, melting the powder into the desired shape. Subsequently, the substrate is lowered by one layer thickness, and the process is repeated until a solid 3D component is

formed. To promote bonding and minimize distortion, the powders used are often preheated to temperatures below their melting point. Additionally, the process typically occurs in an argon-inert gas-filled enclosed chamber to minimize oxidation of reactive powders. Figure 2.1 shows an overview of the L-PBF build chamber.

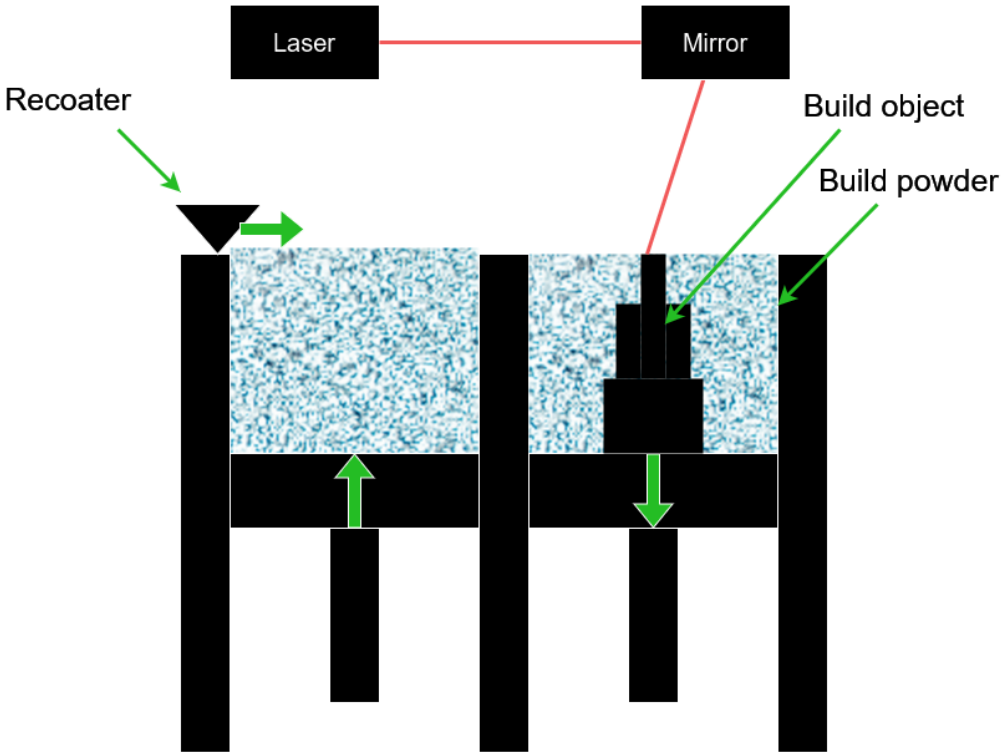


Figure 2.1: An overview of the L-PBF build chamber.

Some of the main in-process parameters for L-PBF include laser power, scanning speed, laser spot size, layer thickness, hatch distance, and build platform pre-heating temperature. These parameters affect the outcome of the print in terms of porosity as mentioned in [7]. For a more detailed view of the parameters and their effect please see [11].

2.1.2 Types of Defects in L-PBF

In L-PBF AM, several types of defects can occur during the printing process [12]. These defects can affect the surface finish, mechanical properties, and overall quality of the manufactured parts. Some of the common types of defects in L-PBF AM are pores and spatters.

- **Pores:** Pores are small voids or cavities within the printed material. They can be caused by trapped gas bubbles, incomplete fusion between powder particles, or insufficient material flow during the printing process. Pores are typically smaller in size compared to other types of voids and may not always be visible to the naked eye.

Porosity: Porosity refers to the overall volume fraction of void spaces within the printed part. It is a measure of how much of the material volume is occupied by pores relative to the total volume of the part. Porosity can be expressed as a percentage and is often used to assess the quality and density of AM parts. Higher porosity levels generally indicate lower material density and reduced mechanical strength.

There are three different types of pores:

Gas Pores: Gas pores are voids formed by trapped gas bubbles within the melted material. These bubbles can originate from various sources, such as entrapped air between powder particles, escaping from the powder material, or vaporization of volatile components. Gas pores are typically small in size and can be evenly distributed throughout the part.

Keyhole Pores: Keyhole pores are larger voids formed by the collapse of vapor-filled cavities within the melt pool. During the L-PBF process, intense laser energy vaporizes the metal at the center of the melt pool, creating a cavity known as a keyhole. If the keyhole collapses before the surrounding metal solidifies, it can leave behind a void in the printed part.

Lack of Fusion Pores: Lack of fusion pores occurs when adjacent layers of melted material fail to bond properly during the printing process. This can happen due to insufficient energy input, inadequate powder spreading, or improper scanning strategies. Lack of fusion pores are typically elongated or irregular in shape and can weaken the mechanical integrity of the part.

Spatters: Spatters are particles that are ejected from the melt pool in L-PBF due to the high thermal energy caused by the laser. They can lead to structural weakness and surface irregularities in a print. Spatters also disturb the printing process by causing blockages and contaminating the build environment which can cause other types of defects such as pores [3].

2.2 Optical Tomography

Optical thermography (OT) [13] images, also known as infrared thermography images, are created using thermal cameras that capture the infrared radiation emitted by objects. These images, or thermograms, reveal the temperature distribution within an object, allowing for the identification of areas with different temperatures.

The process of creating thermograms involves the use of infrared cameras that detect radiation in the long-infrared range of the electromagnetic spectrum (wavelength 9 - 14 μm). This range is chosen because, for all objects with a temperature above absolute zero, it is possible to see infrared radiation, meaning it is possible to see their thermal output without illuminating the object using a light source. The radiation quantity released by an object increases with temperature, which is why thermography allows for the visualization of temperature differences. Objects of high thermal intensity stand out against colder backgrounds, making it easier to identify areas of interest or defects within a part.

In the context of AM, thermographic images can be used to monitor the quality of parts during the manufacturing process. By analyzing the temperature distribution within a part, manufacturers can identify defects such as cracks or voids that may affect the part's performance. This non-destructive testing method is beneficial because it does not alter the part, allowing for adjustments to be made to improve the quality of the part.

2.3 Stream Processing

Stream processing is the process of continuously analyzing, manipulating, and acting upon unbounded data streams in real-time or almost real-time as they are generated. Instead of storing data initially and processing it afterward, it entails processing data as it is being generated. Stream processing is widely employed in many different applications, including real-time analytics, fraud detection, monitoring systems, and Internet of Things (IoT) applications, where prompt insights and actions are needed.

2.3.1 Key Concepts in Stream Processing

- **Data Streams:** Data streams are continuous flows of data that can come from various sources such as sensors, logs, or other operators.
- **Tuples:** The data streams in stream processing consist of *tuples*. A tuple consists of multiple elements, typically including metadata, such as timestamp and size, and the actual payload (the data itself) [14]. The notation of a tuple used in this thesis is $Tuple<metadata, data>$.
- **Operators:** The processes that perform transformations on the data streams. Operators can be *stateless* or *stateful*:
  - **Stateless Operators:** Process each tuple independently, without maintaining any state between different tuples. An example is a *map* operator that transforms each input tuple individually.
  - **Stateful Operators:** Maintain a state that is updated across multiple tuples. These often use a *window* to manage subsets of the data stream. An example is a window operator that calculates a running average over a defined time window, for example, aggregating weather data over one hour to one data point.
- **Job**: The entire data processing application created by the user is referred to as a *job* in stream processing. The job consists of multiple operators that execute the data-processing logic to perform operations on the data stream. These operations can handle a wide range of *data transformations* and *aggregations*, enabling complex processing workflows. These operators are structured as a *directed acyclic graph* (DAG).
- **Task:** A *task* is the basic unit of work that is scheduled and executed by the stream processing system. Each task corresponds to a single parallel instance

of an operator. For example, if the operator has a parallelism of 5, it is divided into 5 different tasks, each executed individually.

- **Directed Acyclic Graph (DAG):** The job is structurally represented by a *directed acyclic graph* where the nodes represent the operators and the edges represent the data streams between them. A DAG has no cycles and thus data in the job only flow in one direction, from upstream operators to downstream ones.

2.3.2 Overview of Apache Flink

Reader's Note: *This section provides an overview of Apache Flink. While the information presented is comprehensive, not all details are essential for understanding the subsequent chapters of this report. This background is crucial for grasping the broader context of stream processing. The most relevant subsections are 2.3.2.3 on Flink Operators (specifically source, map, window, and sink operators) and 2.3.2.4 on State Management.*

Apache Flink [15] is an open-source stream processing framework for distributed systems that supports batch and stream data processing. It includes high-level libraries for Java, Scala, and Python. Unlike other stream processing frameworks, such as Apache Hadoop, which typically operates on processing time, Flink processes data based on event time, ensuring accurate timely analysis. Financial transactions and IoT sensor data requiring correct time semantics are two examples of scenarios where Flink is more useful.

2.3.2.1 Apache Flink Architecture

Apache Flink operates within a distributed cluster environment. At its core, Flink programs are represented as Directed Acyclic Graphs (DAGs), known as dataflow graphs. The dataflow graph serves as the blueprint for job execution, detailing the flow of data and the sequence of operations that need to be performed. This structure allows Flink to execute complex data processing tasks efficiently and reliably.

The two main components of the Flink architecture are a *Job Manager* and multiple *Task Managers*. The Job Manager plays a crucial role by scheduling tasks, managing their states, and recovering from failures. In contrast, Task Managers are responsible for executing the tasks assigned to them by the Job Manager. Each Task Manager handles one or more subtasks, managing their state and communication while targeting efficiency.

Each Task Manager can offer one or more task slots to the job, representing the maximum number of parallel tasks the Task Manager can execute simultaneously. To minimize thread synchronization and buffering overhead, Flink may combine operators into a single task. Furthermore, tasks within the same job can share a task slot to optimize resource utilization. The number of task slots required for a job is determined by the highest parallelism level of any operator in that job. For example, if a job has operators with parallelism levels of 2, 4, and 3, the cluster would need only 4 task slots.

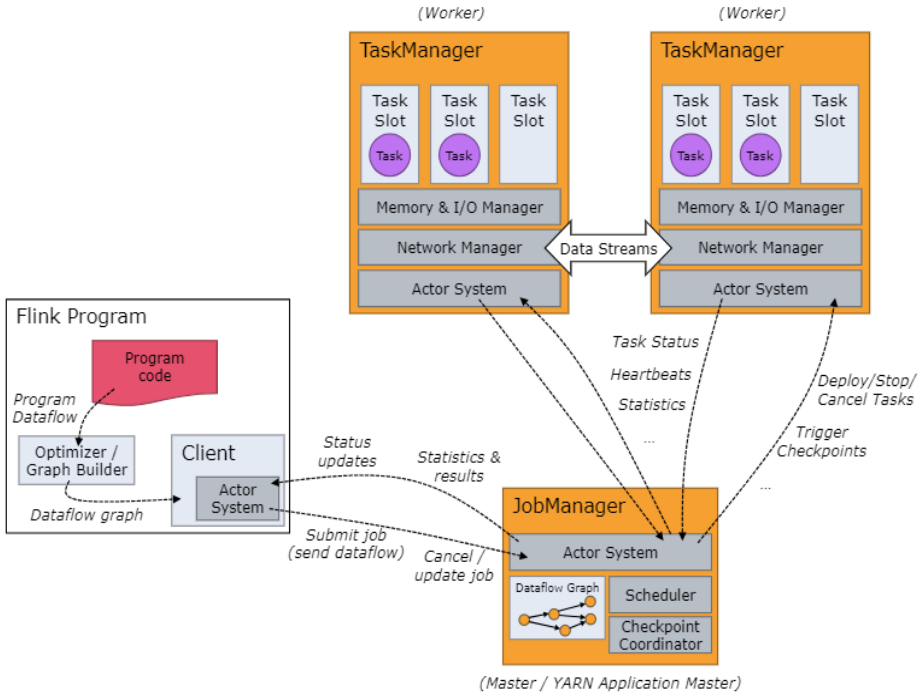


Figure 2.2: An overview of the Apache Flink architecture. Picture taken from [16]

In a high availability configuration, multiple Job Manager instances can be established, with one designated as the active leader and the remaining instances set as standby. These standby Job Managers are prepared to assume the leadership role when the active leader fails.

In addition to the Job Manager and Task Managers, the Flink architecture includes several other key components: the Dispatcher, Resource Manager, Checkpointing, and the Dataflow Graph. The Dispatcher is responsible for handling job submission requests from clients. It manages the Job Manager and provides an interface for job monitoring and management, targeting that job submissions are processed smoothly and efficiently.

The Resource Manager is another vital component, tasked with allocating resources such as memory and CPU for Flink jobs. It interacts with underlying resource management frameworks like YARN, Mesos, or Kubernetes to ensure that the necessary resources are available and properly allocated for job execution [17]–[19].

Fault tolerance in Flink is supported through mechanisms like checkpointing and savepoints. Checkpointing periodically saves the state of the application, enabling recovery in case of failures. Savepoints, on the other hand, are manually triggered, consistent snapshots of the application's state that can be used to restart jobs from a specific point.

2.3.2.2 Apache Flink Key Features

A key feature of Apache Flink is its ability to achieve parallelism, to enable concurrent execution of tasks across distributed computing resources. Flink's parallelism is achieved through its distributed runtime environment, which allows the streaming dataflow to be partitioned and executed in parallel across multiple worker nodes in a cluster. The sample dataflow with Apache Flink parallelism can be seen in Figure 2.3. The degree of parallelism can be configured at various levels, including the operator level, the execution environment level, or the individual task level. This flexibility allows users to fine-tune the parallelism based on their specific workload characteristics, resource availability, and performance requirements.

Flink's state management functionalities provide storage and retrieval of application state, which is essential for constructing stateful stream processing pipelines. With an array of Connectors, Flink integrates with various data sources and sinks, facilitating interoperability with diverse ecosystems.

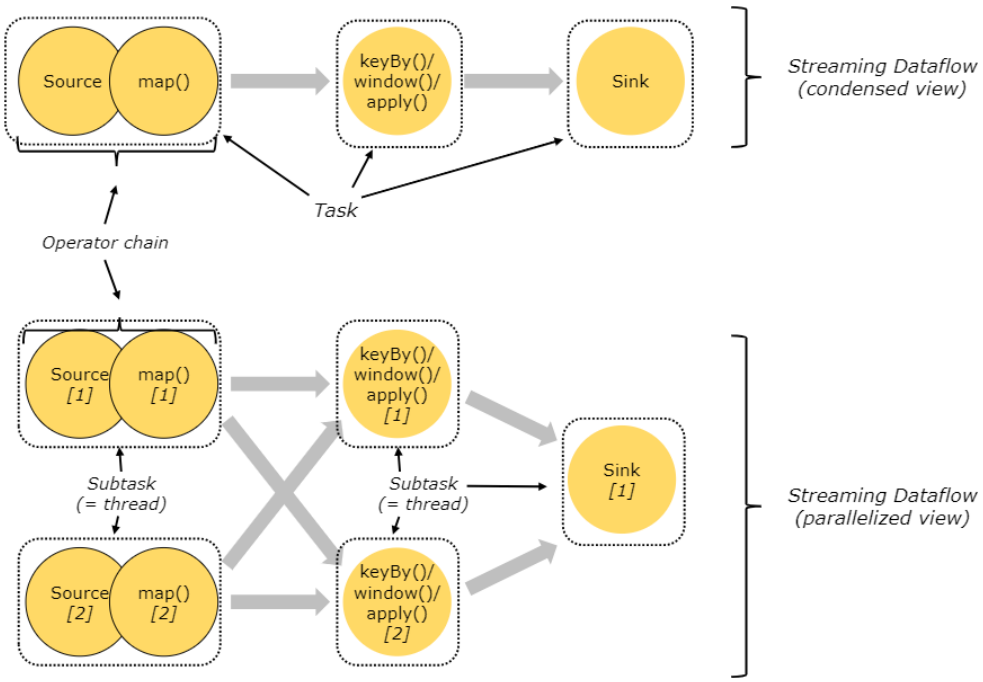


Figure 2.3: A figure demonstrating Apache Flink parallelism. Picture taken from [16]

2.3.2.3 Apache Flink Operators

Apache Flink provides a set of operators for transforming and analyzing data streams. These operators can be combined to build complex data processing pipelines. The Flink operators can be categorized into three types:

- **Source Operator:**

  Source operators are the entry points for data streams in Apache Flink. They ingest data from external sources and emit it into the Flink streaming dataflow

[15]. The source operators in FLink can read the data from multiple data sources such as the local file systems, distributed file systems such as HDFS or S3, sockets, message queues such as Apache Kafka, etc.

- **Transformation Operators:**

  Transformation operators [15] transform one form of the data stream into another form of the data stream by performing some logical operation. Some commonly used transformation operators are

  - **Map Operator:** The map operator takes a DataStream as input and produces a DataStream as output. It independently transforms each element in the data stream. Each element is taken as input and exactly one element is produced as output. The transformation logic is defined by a user-defined function that implements the MapFunction interface.

  - **FlatMap Operator:** The flatMap operator is a versatile transformation operator that takes a DataStream as input and produces a DataStream as output. Unlike the map operator, which emits exactly one element for each input element, the flatMap operator can emit zero, one, or multiple output elements for each input element. This makes it suitable for operations such as splitting elements, filtering out certain elements, or emitting multiple results from a single input. The transformation logic for the flatMap operator is defined by a user-defined function that implements the FlatMapFunction interface.

  - **KeyBy Operator:** The KeyBy operator transformation from a DataStream to a KeyedStream is used to logically partition a data stream into disjoint partitions based on a key. The result is a KeyedStream, where each partition contains elements that share the same key. This partitioning is essential for operations like windowing, aggregations, and stateful processing, where it is crucial to process records with the same key together. Keying a stream is achieved by specifying a key selector function that extracts the key from each element in the stream. Once keyed, Flink ensures that all records with the same key are sent to the same parallel instance of an operator, enabling efficient and accurate processing of keyed operations.

  - **Window Operator:** Window operators in Apache Flink are used to divide a data stream into finite chunks called windows. This transformation is applied to a KeyedStream/DataStream, resulting in a WindowedStream, which allows operations over specific subsets of the data. Windows can be time-driven, like every 30 seconds, or data-driven, like every 100 elements. Flink supports various types of windows, including tumbling windows, which are fixed-size and non-overlapping; sliding windows, which are fixed-size and can overlap; and session windows, which group events based on periods of inactivity. This windowing capability is crucial for performing time-based aggregations, calculations, and other stateful processing tasks on a stream of data. To understand more about

the window operator refer to Figure 2.4.

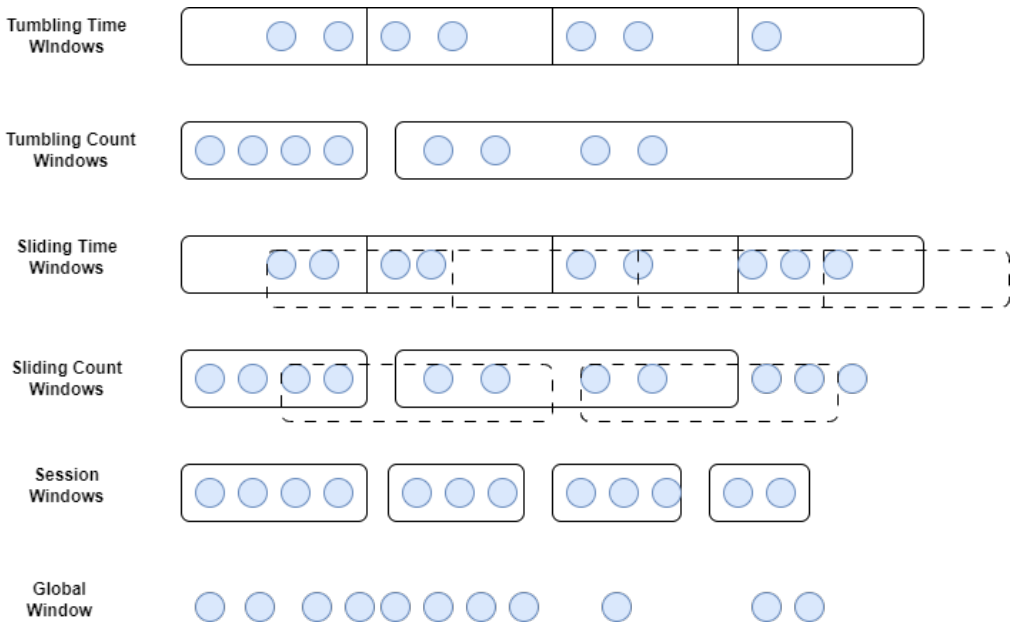


Figure 2.4: A figure demonstrating different types of window operations in Apache Flink. Figure adapted from [20]

– **Join Operator:** Join operator in Apache Flink is used to combine two data streams based on a common key and lie in the same window. Utilizing the Join operator enables users to build complex data processing pipelines integrating information from diverse sources for comprehensive analysis. Flink supports various types of joins, including Window Joins, Tumbling Window Joins, Sliding Window Joins, Session Window Joins, and Interval Joins. These joins allow for flexible data integration strategies, accommodating different time-based and key-based matching criteria. For instance, Window Joins combine elements from two streams that share a common key and fall within the same defined window, while Interval Joins match elements based on relative time intervals. Each join type has specific use cases and behaviors, such as the inner-join semantics of Window Joins or the time-bounded matching of Interval Joins.

- **Sink Operator:**

  The Sink Operator is the endpoint of the Flink streaming pipeline. It is responsible for writing the output of the Flink streaming pipeline to external systems, such as databases, file systems, or message queues. Similar to how the Source Operator can read data from multiple sources, the Sink Operator can write data to various destinations such as local or distributed file systems (e.g., HDFS, S3), relational databases (e.g., JDBC, MySQL, PostgreSQL), messaging systems (e.g., Apache Kafka, RabbitMQ), Elasticsearch for search and analytics, etc [21].

2.3.2.4 State Management in Apache FLink

In Apache Flink, state management is an essential component of stream processing, particularly for stateful operations. The *Map State Descriptor* is one of the primary state abstractions offered by Flink. This descriptor is used to specify a state structured as a map, with each key associated to a value. This is especially useful in situations when a dynamic set of key-value pairs have to be kept.

- **State Descriptor:** A state descriptor in Flink defines the type and attributes of the state that an operator will use. A particular kind of state descriptor for handling map-like state is the Map State Descriptor.
- **Map State:** The Map State allows to store and retrieve values based on keys. New entries can be added, existing ones can be updated, or removed from the state. It is similar to a dictionary or a hash map in other programming languages. Each entry in the map consists of a key and a value.

2.4 Outlier Detection

Detecting anomalies in data is known as outlier detection; that is, finding points of data in a data set that deviates significantly from the values of the other points in that data set. Outlier detection has multiple use cases ranging from credit card fraud detection to finding irregularities in urban traffic [22]. In this thesis, the outlier detection is based on spatial data analysis. In L-PBF an anomalous thermal radiance, i.e. temperature, can lead to porous areas being created [6]. Therefore thermal outliers can be marked as suspicious points when looking for porous areas. In this thesis, the outlier detection method is based on the one used in [11] and the basis of it is to find outliers by comparing their Z-scores to their neighborhood Z-scores using the equation in 2.1, where p is the value of a data point and μ is the mean value of the set (in this work the set is the thermal intensity of a layer) and σ the standard deviation.

$$Z_{score}(p) = \frac{p - \mu}{\sigma} \tag{2.1}$$

The neighborhood size and shape are specified individually and explained in Section

4.3. How much of an outlier a point is depends on the difference in its Z-score and the average Z-score of its neighborhood. A plot of an example output of this method is shown in Figure 2.5.

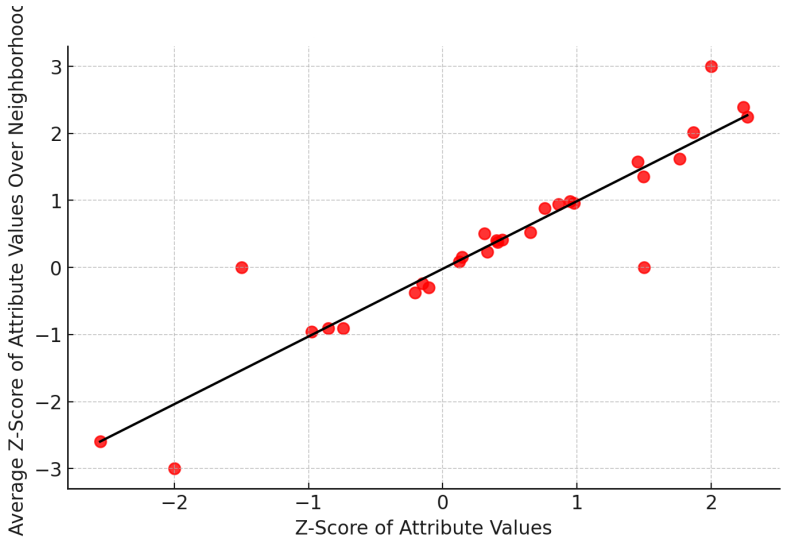


Figure 2.5: A mock plot of data point's Z-score on the x-axis compared to the average Z-score of their neighborhoods on the y-axis, closer to the line means less of an outlier.

2.5 Data-point Clustering and Associated Representations

2.5.1 Clustering of Data-points

In this thesis, to identify suspicious areas within each layer of the OT image, we employed an unsupervised machine learning (ML) technique known as *clustering*. Clustering means to divide a set of points into distinct groups, and clusters, based on their values. The two most widely used clustering algorithms are K-Means and Density-Based Spatial Clustering of Applications with Noise (DBSCAN), each with distinct characteristics [23], [24].

K-Means work by selecting k random data points as *centroids*. The clusters are then initially created by assigning each data point to the closest centroid. After that, the centroids are updated by calculating the mean of each cluster created, and the points are again assigned to their closest centroid. This is done iteratively until the clusters do not change significantly or a maximum number of iterations is reached. The drawback of this method is of course that there needs to be an idea of the number of clusters beforehand. This is why DBSCAN is often used instead.

DBSCAN uses data point density to identify clusters. Unlike K-Means, DBSCAN does not require a predefined number of clusters and can detect clusters of arbitrary shape. DBSCAN is also more suitable for datasets with different densities and

unusual cluster forms since it can effectively label and handle noise points. This capability makes DBSCAN a better fit for our purpose. An image demonstrating the differences between K-Means and DBSCAN clustering algorithms can be seen in Figure 2.6. It clearly shows the need for K-Means to divide the set into a predefined number of clusters even though it might not be the best solution in many cases.

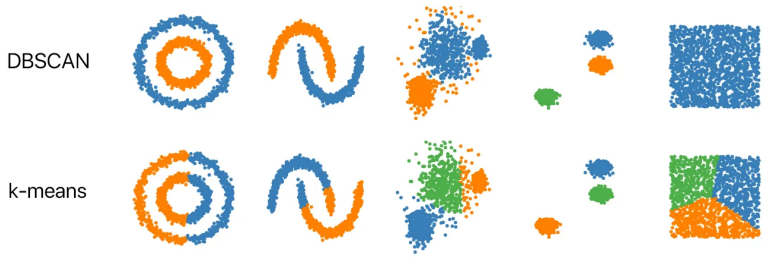


Figure 2.6: Clustering differences between K-Means and DBSCAN. Image is taken from [25].

2.5.2 DBSCAN

DBSCAN is a clustering algorithm first introduced in [24], that is designed to find clusters in a dataset by looking at the density of data points in the set. It has two input parameters ϵ and $minPts$, ϵ is the maximum distance in which the algorithm will look for neighboring points, $minPts$ is the minimum number of points required to lay within distance ϵ from the point at hand for this point to be called a *core point*. If a point does not have $minPts$ neighbors within distance ϵ it is called a *noise point*. The algorithm randomly chooses a point and if it is a core point it starts building a cluster there and expands it incrementally by adding the neighboring core points. This is done until there are no more core points in the distance ϵ. All the noise points within ϵ of a core point in the cluster also get added to the cluster but not their neighbors. If there are still non-clustered core points after a cluster has been built, the algorithm starts a new cluster at a random non-added core point. A pseudo-code for the DBSCAN algorithm can be seen in Algorithm 1. DBSCAN runs with worst-case complexity $\mathcal{O}(n^2)$ but expected complexity $\mathcal{O}(n * log(n))$, with n being the number of points in the dataset. There exists some stream-compliant (incremental) algorithmic implementations with near-linear complexity if the data has spatiotemporal correlations [26] or some approximation is allowed [27], [28].

Algorithm 1 DBSCAN algorithm [24]

Input: A set of points A, ϵ, $minPts$

```
1: for each unvisited node a ∈ A do
2:     N ← Neighbours(a, ε)
3:     if |N| < minPts then
4:         noisePoints ← a
5:         mark a as visited
6:     else
7:         C ← a
8:         for each unvisited node a′ ∈ N do
9:             N ← N/a′
10:            if a′ is not visited then
11:                mark a′ as visited
12:                N′ ← Neighbours(a′, ε)
13:                if |N′| < minPts then
14:                    N ← N ∪ N′
15:                end if
16:            end if
17:            if a′ is not yet a member of any cluster then
18:                C ← C ∪ a′
19:            end if
20:        end for
21:    end if
22: end for
```

2.5.3 Axis-Aligned Bounding Boxes

When given a set of points it can be beneficial to have a geometric shape to enclose the points to give an approximate description of the shape, size, and density of the points. This is what is called a geometric summary and it can simplify calculations on data points as utilized in [28] to do an approximate parallel implementation of DBSCAN. Creating a geometric summary can vary in complexity depending on the specific method utilized and how tight of a fit around the points is needed. A simple geometric summary method is called *axis aligned bounding boxes (AABB)*, and it works by finding the boundary points of a dataset, i.e. points that have the maximum or minimum value in any dimension [29]. For example, for a data set consisting of two dimensions we would have a x_{min}, y_{min} as well as a x_{max} and a y_{max} that gives a set of points of which a bounding box will be created. An illustration of this can be seen in Figure 2.7. An advantage of AABB is its simplicity and relatively low complexity, linear search means $\mathcal{O}(n)$. A disadvantage is that it might yield a bad fit for data points that do not cover a rectangular area resulting in empty space in the geometric summary.

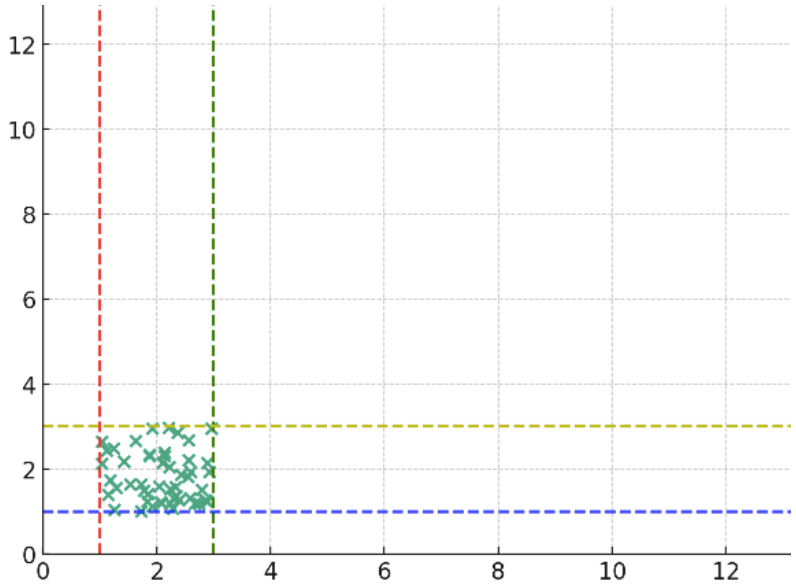


Figure 2.7: An example of Axis Aligned Bounding Boxes in 2D where the lines indicate the max and min value of the dataset and their intersections indicate the corners of the boxes. Red line = x_{min}, green line = x_{max}, blue line = y_{min} and yellow line = y_{max}.

2.6 Classification Metrics

A classification ML model converts the input of the model into an estimate of that input belonging to a certain category [23]. A classification ML model could for example take as input a picture and output whether that picture has a dog or a cat in it. A binary classification is simply a classification model that only has two possible outputs. For such a model the evaluation of the outcome ends up in one of four classes:

- True Positive (TP): An instance the model classifies as positive and is positive.
- False Positive (FP): An instance that the model classifies to be positive but is negative.
- True Negative (TN): An instance that the model classifies as negative and is negative.
- False Negative (FN): An instance that the model classifies as negative but is positive.

To evaluate the performance of these types of models there are three metrics: Accuracy, Precision, and Recall. they are calculated as follows.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

3

Related Work

This section contains a summary of related literature that discusses Quality Monitoring in AM, utilizing streaming techniques, and incremental clustering methods.

There are multiple methods to classify defects in AM using different types of ground truth data and ML. Most existing works focus on the accuracy of classification within individual layers (intra-layer) rather than examining how defects span across multiple layers (inter-layer) or the time required to analyze these layers. Additionally, there is ongoing research on creating the outline of an entire Stream Processing Framework to be able to integrate in-situ monitoring online and live.

3.1 In-Situ Defect Detection in L-PBF

Grasso et al. [5] provide a comprehensive review of in-situ measurement and monitoring methods for metal powder bed fusion (PBF) processes. The authors classify the in-situ methods into five levels based on measurement scale and type, ranging from machine state monitoring to subsurface imaging. In this thesis, thermal images will be used which aligns with their Level 2 classification, which involves measurements taken during laser scanning of each layer. The authors emphasize the critical link between in-situ measured quantities and the quality of the final part, highlighting the influence of process parameters on defect formation mechanisms. This aligns closely with our approach which aims to use thermal signatures to identify potential defects. The review also underscores the growing research interest in developing novel in-situ measurement, monitoring, and control solutions, which resonates with our work on low-latency defect detection using thermal imaging. The work done in [5] serves as an important reference point for understanding the current state-of-the-art in-situ monitoring for metal powder bed fusion and helps position our work within the broader landscape of ongoing research efforts to advance zero-defect additive manufacturing capabilities.

Mohr et al. [7] presented a study on in-situ defect detection in L-PBF processes using thermography (thermal radiation) and optical tomography, with a comparison to X-ray CT (CT). The research aims to enhance process understanding and quality assurance in AM by integrating multiple sensing methods and advanced data analysis techniques. They show the effects different laser parameters have on the print quality and that the thermal signature of the powder during printing can indicate porous areas. The findings of [7] motivate the hypothesis of the work in this thesis

which is based on the idea that those thermal outliers can indicate regions of interest and quantify their size and location for further inspection.

Estalaki et al. [30] utilize in-situ thermal imaging to capture the thermal history of voxels during the L-PBF process, providing direct insights into temperature variations that may lead to defect formation. They extract two key thermal features, namely the time above the apparent melting threshold (τ) and the maximum radiance (T_{max}) of each voxel, from the thermal imaging data. These features are then used as input variables for several classifications ML models, such as k-nearest neighbors and decision trees. Additionally, the thermal features of each neighboring voxel, are considered as inputs to improve prediction accuracy. This preprocessing step takes into account the thermal history around each voxel, considering heat transfer effects in defect formation while excluding boundary pixels. Our thesis builds on the premise created by [30] and others that ML can be used to interpret data from printers. We do however do this using unsupervised ML.

Sievers [11] focuses on predicting porosity in Laser Powder Bed Fusion (L-PBF) using thermal data extracted from optical tomography (OT) images. The thesis emphasizes the importance of thermal outliers, which are pixels with temperatures significantly different from their neighboring pixels. These outliers are identified using spatial statistics algorithms, specifically the Moran Scatter plot while excluding boundary pixels. The identified thermal outliers are then used as input to trained K-Nearest Neighbor (KNN) and Decision Tree ML models to classify whether particular objects are porous or not. Our thesis uses similar methods for detecting outliers as [11], but diverges in several key aspects. We employ a different classification method, place more emphasis on timeliness in processing, and extend the analysis to examine defects across multiple layers. This approach allows us to explore new dimensions in defect detection while leveraging established outlier identification techniques.

3.2 Data Driven Stream Processing Framework for AM

Gulisano et al. [14] propose a framework called STRATA which supports online monitoring in AM by leveraging stream processing, with a specific focus on L-PBF printing. STRATA offers tailored APIs and processing paradigms for AM processes, facilitating the development of custom monitoring pipelines. The framework is designed to provide scalable, low-latency, and high-throughput solutions for data-driven AM processes. By utilizing common processing paradigms such as stream processing and key-value stores, STRATA ensures scalable data analysis and portability. The authors demonstrate the framework's functionality through a real-world use case for online monitoring of PBF-LB printing processes. Their evaluation involves clustering cold and warm regions both within and across layers using DBSCAN. In our work, we use a similar approach to find regions of interest in that we use DBSCAN but we do this without the need for a specific temperature threshold and also give an overview of the geometry of the clusters. Our work is however complementary to

[14] as the pipeline can be integrated into such a framework to aggregate the data from the printer.

3.3 Incremental Clustering for LIDAR Data Streams

There are several methods for creating specific implementations of DBSCAN to cluster *LIDAR* data. A *LIDAR* works similarly to short-range radar and builds up a 3D image of its surroundings by continuously sending out laser beams and measuring the time until they come back. By measuring time it knows the distance to any object nearby and due to its rotation, it gets a 360-degree view of its surroundings.

H. Najdataei et al. [26] introduce an algorithm called Lisco to do Euclidian distance-based clustering (like DBSCAN) of point clouds coming from a LIDAR sensor continuously. The algorithm takes advantage of the inherent nature of data coming from a LIDAR by creating a "ϵ-neighborhood" around a received point and checking if future points land within that neighborhood. As the new points will lay in front of the points already looked like due to the rotation of the LIDAR sensor it is an efficient way to cluster the data continuously. Because the data is already ordered when it is delivered there is no need to put it in a spatial data structure to do an optimized search of points. [26] also works with data streams other than from LIDARs as long as the data points are correlated in time and space.

A. Keramatin et al. [28] introduce a set of parallel implementations of DBSCAN called PARMA-CC that divides a data set into different regions and assigns each region a thread to run DBSCAN on. The clusters of different regions were then merged by comparing a geometric summary of their data points to see if they overlap. The implementation was tested on LIDAR data and showed good scalability.

Also in A. Keramatin et al. [27], a method called MAD-C is created to utilize data summaries of point clouds to are used to fuse point clouds coming from multiple LIDAR sensors into one object to create a merged view of the surroundings of these LIDAR sensors.

Our method is inspired by the aforementioned works, in the sense that geometric summaries are combined, through comparisons of data objects that are spatiotemporally correlated. The next chapter describes the methods in detail, along with implementation aspects in the Flink stream processing system.

4

Method

This chapter provides an overview of the data pipeline, detailing the pseudo-code for each operator within it. It describes the input and output data streams for each operator, as well as the parameters they require to function. Figure 4.1 illustrates the directed acyclic graph (DAG) of the data pipeline, showing the order in which the operators are arranged. As discussed in Chapter 3 these operators work with data ordered by the layers of objects to create summaries of clusters across layers. All the operators are built using Apache Flink and key concepts in that framework, elaborated on in Section 2.3.

4.1 Data Preprocessing

A camera mounted above the build area captures OT images of the entire build plate after each layer is printed. As multiple objects may be printed simultaneously in a single printing job, individual monitoring of each object is necessary to assess their respective build quality. This requires preprocessing the captured thermal images, to crop the region of interest (ROI) for each object. The source operator can then send in one object at a time to the pipeline, allowing the rest of the pipeline to only look the layer number of the data it receives. These cropped images are then fed to our data stream pipeline.

4.2 File Monitoring Operator

The File Monitoring Operator is the source operator of our data stream pipeline. It reads the images from the disk into a 2D array, $image\_data[][]$. It then outputs the array along with the Layer ID (L_{id}) and a time stamp (ts).

The timestamp (ts) is particularly significant as it is forwarded through the data stream pipeline to the last operator. This is used for making sure in-order execution of the operators. This also allows for the computation of latency, providing a measure of the time taken to analyze each image from its detection by the File Monitoring Operator to the completion of processing by the final operator. This end-to-end latency measurement is crucial for evaluating the performance and efficiency of the data processing workflow.

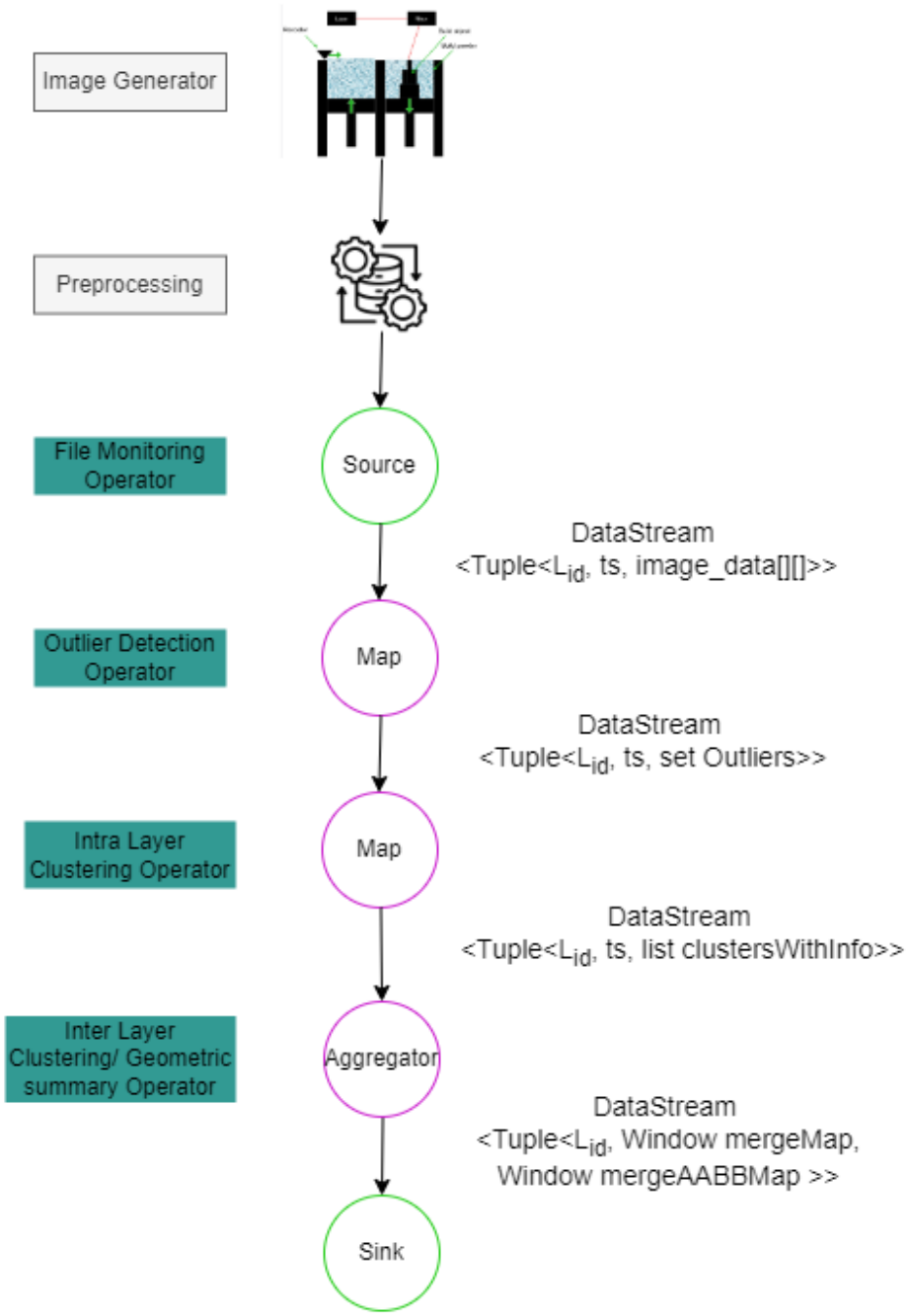


Figure 4.1: The outline of the stream pipeline shown as a directed acyclic graph.

4.3 Outlier Detection Operator

The outlier detection operator is a stateless map operator that identifies outliers in thermal images, which is the first step in the process of detecting suspicious areas. The algorithm to find outliers in this thesis adopts the approach proposed in [11]. The neighborhood shape used to calculate the average Z-score of any point in the dataset is as depicted in Figure 4.2. The Z-score is a measurement of how many

standard deviations a data point is from the mean, as covered in Section 2.4. The algorithm used to find the outliers of an image is presented using pseudo-code in Algorithm 2. As mentioned in Section 2.4 the algorithm is based on spatial data analysis and uses the difference in Z-score compared to the neighborhood average to calculate how much of an outlier a pixel is. The algorithm takes an input tuple containing L_{id}, $image\_data[][]$, and ts. The algorithm also requires two input parameters: *neighborhood size (η)* and *percent of outliers (ξ)*. η is the size of the neighborhood used for calculating average Z-score and ξ is how many percent of pixels are expected to be outliers. These two variables will be varied during evaluation to see the effect on the results. It starts by finding the Z-score of each pixel/data point in $image\_data$ and storing it along with the position x and y of the pixel. After finding the Z-score it creates an average Z-score value using the Z-scores of the pixels surrounding a point as depicted in Figure 4.2 and stores it in a similar manner for each pixel. Lastly, it calculates the difference between a point's Z-score, $a.z_{xy}$, and the average Z-score, $b.z_{xy_{avg}}$, and puts it in a set D along with the coordinates x and y. It then sorts D by that distance, picks the top ξ% of D, and puts the corresponding x and y into the set $Outliers$. The operator then outputs a tuple containing L_{id}, ts, and $Outliers$ into the data stream. This results in only looking at hot spots, as cold spots will have a negative difference and end up at the end of D. If cold spots are to be detected as well D could simply be ordered by the absolute value of the difference instead.

Algorithm 2 Outlier detection(η, ξ)

Input: DataStream Tuple<L_{id}, ts, $image\_data[][]$>
Output: DataStream Tuple<L_{id}, ts, set $Outliers$>
Parameters: Neighborhood size η, Percent of outliers ξ

1: Let t denote the input tuple;
2: For each pixel at coordinates (x, y) in $t.image\_data$, compute the Z-score z_{xy} and store it in a set z_{scores} as vector of the form (z_{xy}, x, y);
3: For each pixel at coordinates (x, y) in $t.image\_data$, compute the average Z-score $z_{xy_{avg}}$ of neighborhood (size η) and store it in a set $z_{neighborhood}$ as vector of the form $(z_{xy_{avg}}, x, y)$;
4: **for** $i \leftarrow 0$ **to** length(z_{scores}) - 1 **do**
5: Let a be $z_{scores}[i]$;
6: Let b be $z_{neighborhood}[i]$;
7: diff = $a.z_{xy} - b.z_{xy_{avg}}$;
8: Store vector of type (diff, x, y) in set D
9: **end for**
10: Sort D in descending order by diff;
11: Take the top ξ% of D and put the corresponding coordinates (x, y) into the set $Outliers$;
12: Output(<$t.L_{id}$, $t.ts$, $Outliers$>)

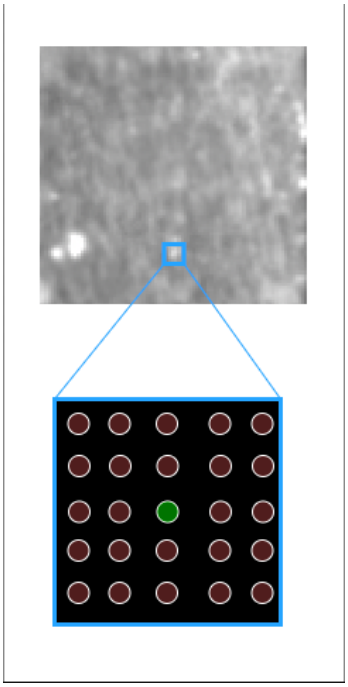


Figure 4.2: The neighborhood for calculation of the average Z-score of a point. The green point is the point of interest and the brown ones are the neighborhood pixels. In this example, the neighborhood size is 5, but multiple sizes will be tested in the thesis.

4.4 Intra-layer Clustering Operator

The intra-layer clustering operator is a stateless map operator that performs clustering on the detected outliers in each layer using the DBSCAN algorithm, to create a set of clusters of suspicious areas. The intra-layer clustering operator also creates an axis-aligned bounding box (AABB) of two dimensions for each cluster, i.e. a summary for each cluster in the form of a rectangle.

The outline of the intra-layer clustering operator is presented in Algorithm 3. The algorithm takes input tuples containing L_{id}, a set of outliers, and ts. The algorithm also requires two input parameters: Epsilon (ϵ), and minimum points ($minPts$) for the DBSCAN clustering, see Algorithm 1 in Section 2.5.2.

The operator starts by finding clusters using the $dbscan.cluster()$ method which outputs one list for each cluster found and saves them in the list $clusters$. Each cluster is a list of $DoublePoints$, and a $DoublePoint$ is just a pair of doubles, one for the x coordinate and one for the y coordinate for each point in the cluster. After clustering, the operator puts the information of each cluster into a data class called $clusterInfo$, defined in the algorithm. The operator iterates over each cluster (c_n) in $clusters$ and calculates the AABB using the $getAABB()$ function, which returns a class containing four doubles: $minX$, $maxX$, $minY$, $maxY$ for each cluster. Additionally, c_n is assigned a label (c_{lab}) using a concatenation of L_{id} and $clusterIndex$ and the coordinates of the points in the cluster, the content of c_n, is put into $clusterPts$. It

then adds the $clusterInfo$ and its data to the list $clustersWithInfo$. Finally, the operator outputs a tuple containing L_{id}, ts, and $clustersWithInfo$ into the data stream.

Outliers that were found in the previous step that do not have other outliers in close proximity will not be part of any cluster. This means that outliers that are isolated bright spots will be excluded from further analysis, thus ignoring stochastic changes in temperature that do not affect the quality of the object.

An illustration of the outcome of the intra-layer clustering for one layer can be seen in Figure 4.3.

Algorithm 3 Intra-layer Clustering Operator(ϵ, $minPts$)

Input: DataStream: Tuple<L_{id}, ts, set $Outliers$>
Output: DataStream: Tuple<L_{id}, ts, list $clustersWithInfo$<$clusterInfo$>>
Parameters: ϵ, $minPts$

1: let t denote the input tuple
2: $clusters = dbscan.cluster(t.Outliers, \epsilon, minPts)$; //Store each cluster as a list of coordinates in the list $clusters$
3: **class** clusterInfo
4: **String** c_{lab}; // The label of the cluster
5: **AABB** aabb; // The geometric summary of the cluster
6: **List<DoublePoints>** clusterPts; // A list of the coordinates of the points
7: **end class**
8: $clustersWithInfo$ = new $List$<$clusterInfo$>; //create a list to hold the clusters and their data
9: **for** c_n in $clusters$ **do**
10: ci = new $clusterInfo$;
11: $ci.aabb = getAABB(c_n)$;
12: $ci.c_{lab} = (L_{id}, n)$;
13: $ci.clusterPts = c_n$;
14: $clustersWithInfo.add(ci)$;
15: **end for**
16: Output(<$t.L_{id}$, $t.ts$, $clustersWithInfo$>)

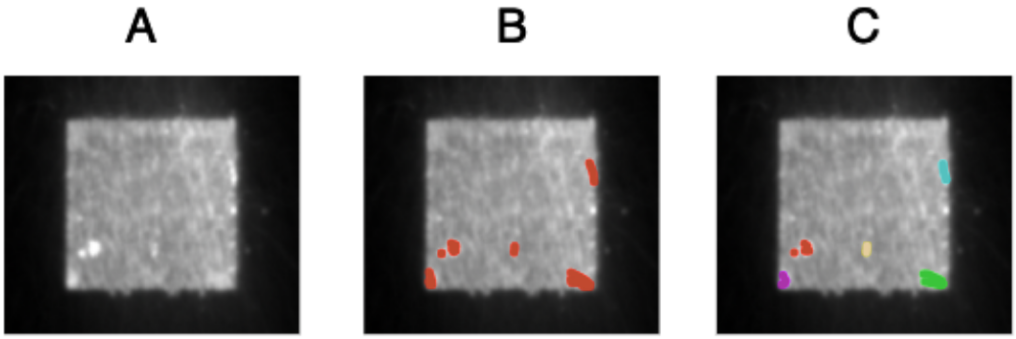


Figure 4.3: A sketch of what the outlier detection and clustering does. A to B shows outlier detection, finding which pixels are outliers, and B to C clusters these outliers to find regions of interest.

4.5 Inter-layer Clustering Operator

The inter-layer clustering operator is a stateful aggregate operator that finds a set of clusters of suspicious areas across the layers. It performs incremental clustering by comparing and merging clusters from the current layer with those from the previous layers.

The outline of the inter-layer clustering operator is illustrated in Algorithm 4. It starts by initializing windows $mergeMap$ and $mergeAABBMap$ of size W. $mergeMap$ is used to keep the state of which **intra**-layer clusters belong to an **inter**-layer cluster and $mergeAABBMap$ is used to keep track of the geometric summaries of the inter-layer clusters and is expanded on in Section 4.6. The two windows are based on a MapState structure, as described in Section 2.3.2.4, and they are represented by a hashtable with a cluster label as a key and a linked list of cluster labels as values for $mergeMap$. An illustration of $mergeMap$ can be seen in Figure 4.5. After initializing the windows, a global variable, $prev\_t$, is initialized to keep track of the tuple belonging to the previous layer to enable incremental construction of inter-layer clusters by comparing the tuples of two adjacent layers.

The operator then moves on to check if $prev\_t$ is "none" in which case it sets it to the just received tuple and does nothing more. When the next tuple then arrives, $prev\_t$ will not be "none" and a second variable $curr\_t$ will be created to hold the content of the tuple corresponding to the current layer. To compare the clusters of two adjacent layers, the operator iterates over each entry in the list $clustersWithInfo$ in $curr\_t$ and each entry in $clustersWithInfo$ in $prev\_t$. The operator checks if the axis-aligned bounding boxes (AABBs) of the entries ci_n and ci_{n-1} overlap, considering an epsilon $\frac{\epsilon}{2}$ boundary extension [1]. An example of AABBs with $\frac{\epsilon}{2}$ boundary extension can be seen in Figure 4.4. If the AABBs overlap, the operator iterates through each point p_i in $ci_n.clusterPts$ (the list of coordinates for the points

[1]Since both AABBs will have an extension, only $\frac{\epsilon}{2}$ is needed as their extension will add together to become ϵ

of ci_n) and each point p_j in $c_{n-1}.clusterPts$. If any pair of such points p_i and p_j are within ϵ distance, then the clusters are merged by calling the *mergeMap.add()* function with parameters $ci_n.c_{lab}$ (the cluster label of the current layer cluster), and $ci_{n-1}.c_{lab}$. If the two AABBs do not overlap we can be sure that no points are within ϵ distance and continue with the next entry.

The outline of $mergeMap.add()$ can be seen in Algorithm 5. The operator checks if the cluster label ($c_{n-1_{lab}}$) already exists in $mergeMap$. If the key exists, the operator fetches the associated linked list of cluster labels and adds the current layer cluster label $c_{n_{lab}}$ to the list. If the key does not exist, the operator creates and initializes a new linked list, adds the previous layer cluster label $c_{n-1_{lab}}$ and the current layer cluster label $c_{n_{lab}}$ to the linked list, and creates a new entry in $mergeMap$ with the key of the previous layer cluster label $c_{n-1_{lab}}$ and the new linked list. The appearance of an entry in $mergeMap$ can be seen in Figure 4.5.

After all clusters of the two adjacent layers have been checked, the keys of $mergeMap$ and $mergeAABBMap$ are updated to represent the most recently added cluster in each entry. This could not be done before because clusters in the current layer need to be able to see which clusters in the previous layer already belong to an inter-layer cluster.

Finally, the operator outputs a data stream containing L_{id}-WindowSize (the first layer of this window), $mergeMap$, and
$mergeAABBMap$ when the windows reaches their maximum size W.

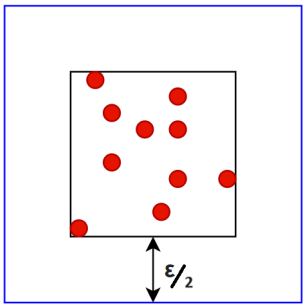


Figure 4.4: An image showing the difference between a regular AABB and an epsilon-adjusted AABB. Black is the regular AABB and blue is the epsilon-adjusted one.

4.6 Geometric Summary

Expanding on [11] we do a geometric summary to get a set of locations representing the clusters found, these are the areas that might contain defects. This geometric summary will describe the shape and size of a cluster. Doing this incrementally allows us to quantify the size of possible defects as the object is being produced and get a 3D location of defects (several layers). The geometric summary will be done by finding the max and min coordinates for x, y, and z of a cluster, thus creating an axis-aligned bounding box (AABB) of the cluster.

Algorithm 4 Inter-layer clustering operator algorithm

Input: Data Stream: Tuple<L_{id}, ts, list $clustersWithInfo$<$clusterInfo$>>
Output: DataStream: Tuple<L_{id}-$WindowSize$+1, $mergeMap$, $mergeAABBMap$>

```
1: Init window mergeMap to keep track of inter-layer clusters;
2: Init window mergeAABBMap to keep track of geometric summaries of
3: inter-layer clusters;
4: Init global variable prev_t to none; //Init mergeMap, mergeAABBMap and
   prev_t only on first tuple of new window
5: if previousLayer == none then
6:     Set prev_t to the received tuple; //If prev_t has no value yet
7: else
8:     Create variable curr_t and set to the received tuple; //Check if any
        clusters in the two adjacent tuples overlap
9:     for  each ci_n in curr_t.clusterWithInfo and
10:        ci_{n-1} in prev_t.clusterWithInfo,
11:        such that AABBOverlap(ci_n.aabb, ci_{n-1}.aabb)  do
12:        for each pair of points p_i in ci_n.clusterPts and
13:            p_j in ci_{n-1}.clusterPts such that |p_i − p_j| < ε do
14:            mergeMap.add(ci_n.c_lab, ci_{n-1}.c_lab);
15:            mergeAABBMap.add(ci_n, ci_{n-1});
16:        end for
17:    end for
18:    Set prev_t equal to curr_t;
19:    Update mergeMap and mergeAABBMap to have the most recently
20:    added entries as keys;
21: end if
22: if mergeMap is full then
23:    Output (<curr_t.L_id - mergeMap.size+1, mergeMap, mergeAABBMap>);
24: end if
```

The geometric summary of an inter-layer cluster will be created and organized in a similar way as the inter-layer clusters themself are in $mergeMap$. There will be a window of size W based on Map State/hash table, called $mergeAABBmap$, with the AABBs of each cluster as values and the label of the latest added cluster as a key. It has a function similar to $mergeMap.add()$ but the value is an AABB instead of a linked list of clusters and for each newly added cluster it needs to check if there are new $maxX$, $minX$, $maxY$, or $minY$. $maxZ$ will always be the most recently added cluster as it has the highest layer number. Figure 4.6 is an example of how the geometric summary of clusters could look like.

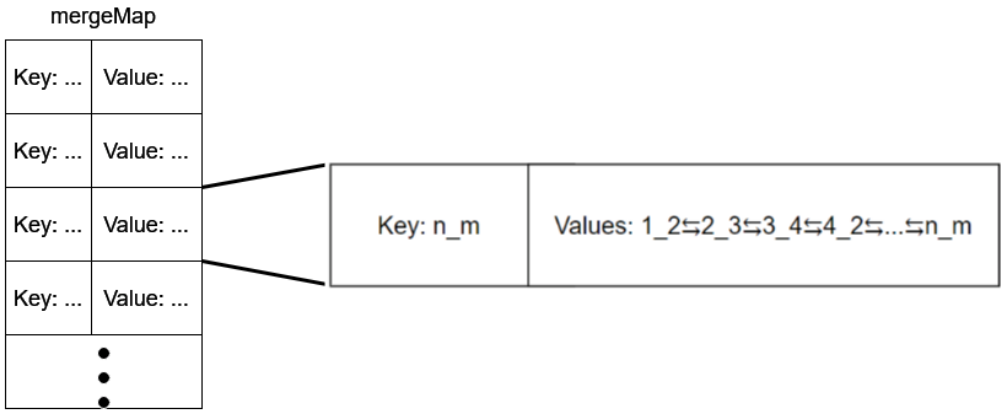


Figure 4.5: What mergeMap and its entries look like, the key is the last cluster label and the values are a linked list of cluster labels. n is the layer number and m is the cluster number.

Algorithm 5 mergeMap.add()

Input: Two cluster labels $c_{n_{lab}}$, $c_{n-1_{lab}}$
Output: None

```
1: //Check if if mergeMap already contains a key with c_{n-1_lab}
2: if mergeMap.contains(c_{n-1_lab}) then
3:     LinkedList<String> values = mergeMap.get(c_{n-1_lab});
4:     //Key already exists, fetch the associated linked list and add the c_{n_lab}
5:     values.add(c_{n_lab})
6: else
7:     //Key does not exist, create a new entry with c_{n-1_lab} and c_{n_lab} as values
8:     LinkedList<String> newList = new LinkedList<String>();
9:     newList.add(c_{n-1_lab});
10:    newList.add(c_{n_lab});
11:    //Set c_{n-1_lab} as the new key
12:    mergeMap.put(c_{n-1_lab}, newList);
13: end if
```

4.7 Sink Operator

The Sink operator in the data pipeline serves as the final stage for processing and storing the analyzed data. It receives a data stream containing tuples with L_{id}-$WindowSize$+1, $mergeMap$, and $mergeAABBMap$ from the inter-layer clustering operator.

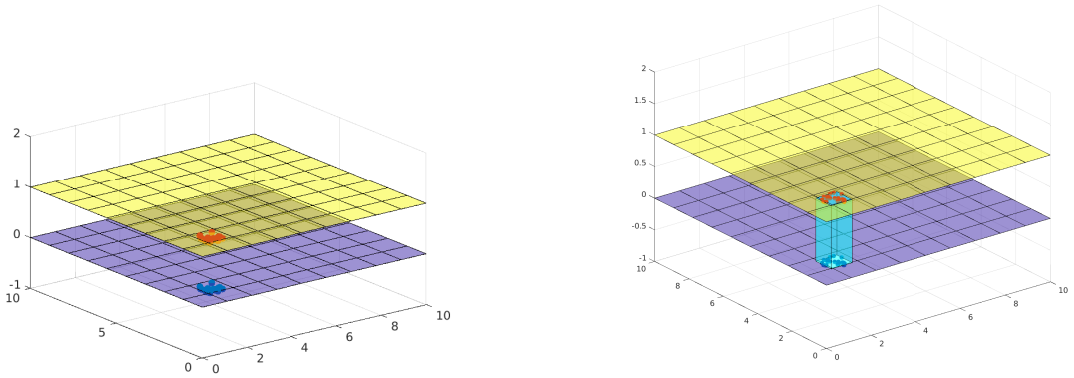


Figure 4.6: Images showing an example geometric summary, the image on the right shows clusters found in two different layers, and the image on the left is the geometric summary of those two clusters.

4.8 Time Complexity of Operators

Outlier Detection The complexity of the outlier detection operator grows with $n * \eta$ where n is the number of pixels in the image of each layer and η is the neighborhood size. For the asymptotic complexity η is constant in relation to n and the worst-case complexity then is linear with n, $\mathcal{O}(n)$ [31]. [2]

Clustering The worst-case complexity for clustering using DBSCAN is $\mathcal{O}(o^2)$ where o is the number of outliers given to the function. For finding the axis-aligned bounding box of each cluster it is $\mathcal{O}(c)$ to find the minimum and maximum of x and y, where c is the total number of points belonging to a cluster. So the total complexity of the clustering operator is still $\mathcal{O}(o^2)$

Inter-layer Clustering The complexity of Algorithm 4 is in the worst case, meaning every cluster overlap, $\mathcal{O}(c{*}p^2)$ where c is the expected number of pairs of clusters that overlap in two adjacent layers and p is the expected number of points in each cluster. The window $mergeMap$ responsible for keeping track of the inter-layer clusters is based on a hash table and has a worst-case complexity for running $contains()$ of $\mathcal{O}(W)$ where W is the size of the window: all inter-layer clusters and their associated labels. The complexities are the same for the geometric summaries window. So summarizing the total complexity of the incremental clustering operator depends on the size of the clusters of an individual layer and how many span across layers, but a fair assumption is there are more points in the clusters of two layers than there are entries in the inter-layer cluster hash table. Because of this, the worst-case complexity should be $\mathcal{O}(c * p^2)$.

[2] $\mathcal{O}()$ is a way to describe the upper bound of the algorithm's asymptotic complexity giving an idea of how it scales with the data.

5

Evaluation

5.1 Goal

The main goal of the evaluation was to test the accuracy of our method in finding defects in AM. We wanted to test how well the clusters that we created using the method overlapped with defects in an evaluation data set. We wanted to test this for both intra-layer clusters as well as inter-layer clusters. For the inter-layer clusters we also wanted to look at how their size changed depending on the quality of the object. For the timeliness, we wanted to look at the latency on a per-image basis, i.e. how many milliseconds it takes to process one image and compare it to other relevant work such as [11]. Another goal was to evaluate the effect of the input parameters given to the operators and their effect on both timeliness and accuracy.

5.2 Dataset Description

This section contains information about the object details, sensor data used as input, and evaluation data set used to evaluate the result obtained.

5.2.1 Optical Tomography Images

We used optical tomography images generated by a sensor/camera with a band-pass filter, fixed on top of the powder bed. The powder bed of the AM machine was of the dimensions 16000 mm X 16000 mm, and the field of view of the camera was 0.125 μm X 0.125 μm per pixel. The image obtained by the camera had the dimensions 2000x2000 pixels.

The camera was imaging a powder bed building 64 cylindrical objects. The objects were after printing put in a lathe and machined to look like a dumbbell. The dimensions of objects after machining can be seen in Figure 5.3. The 64 objects are categorized into three classes of objects A, B, and C printed with different parameters. A has the worst parameters, B the second worst, and C the best. The layout of 64 objects that were printed on the build plate can be seen in Figure 5.1.

Each object printed had a diameter of 11 mm at the top and bottom layers of the object but only around 6 mm at the machined part. In this thesis, the method is evaluated only on the machined part of the object, from layer number 291 to layer

number 540. The portion of the object being analyzed is highlighted in blue in Figure 5.3.

An example of the single layer of the optical tomography image captured by the camera can be seen in Figure 5.2.

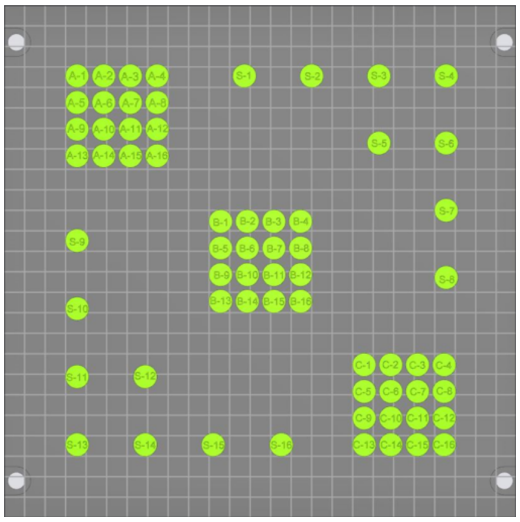


Figure 5.1: A layout print on the build plate. Image generated by the PMAM groups lab[1] and used with permission [32].

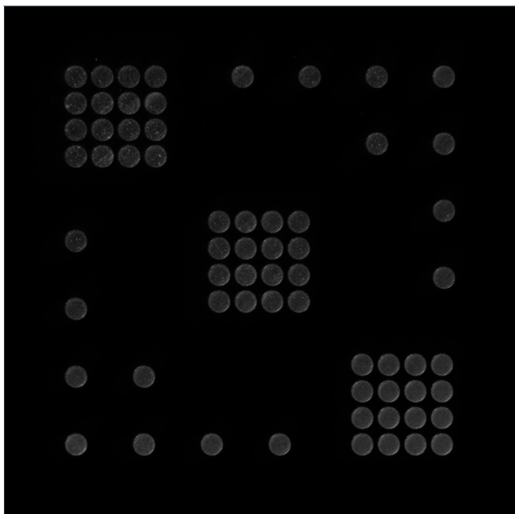


Figure 5.2: A Single layer of the Optical Tomography image. Image generated by the PMAM groups lab[1] and used with permission [32].

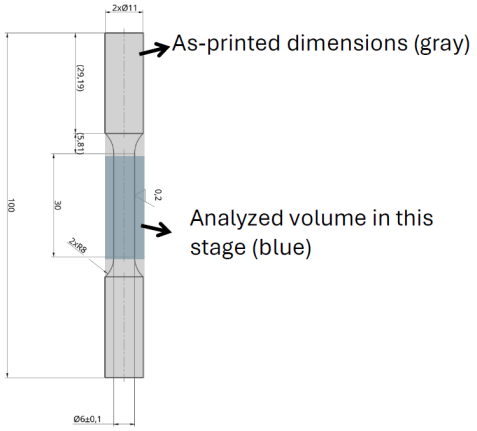


Figure 5.3: Object geometry dimension. Image generated by the PMAM groups lab[1] and used with permission [32]

5.2.2 Evaluation Dataset

In general, the ground truth most useful for quality monitoring in AM is X-ray images. Such data could not be attained for the evaluation done in this thesis,

[1] https://www.chalmers.se/en/departments/ims/research/materials-and-manufacture/powder-metallurgy-and-additive-manufacturing/

but we used data created using the methodology in [33] for locating spatters. The spatters are the particles that jump and fall in some other area of the object while melting the metal powder in a selective area to create the object. Some parameters for the data used were: airflow moved from top to bottom, layer thickness was 120 μm and the layout of the objects is as shown in Figure 5.1.

To locate the spatters the authors of [33] proposed to use Gaussian filtering to "smooth" the image. This process slightly blurs the image to remove high-frequency noise while maintaining the underlying structure. The smoothed images are then processed through the Laplacian filter, which detects edges and features by identifying regions of rapid intensity change. Since spatters exhibit very high temperatures, they create areas of rapid intensity variation where they have fallen. The output of the Laplacian filter, which highlights these rapid intensity changes, indicates the spatter locations. The hypothesis in [33] was that more areas of rapid intensity change would result in a weaker object. In our thesis, the output from the method mentioned above ([33]) for locating spatters was used as an evaluation set to assess the accuracy of clusters/suspicious areas found. This evaluation set had been verified by stress-testing the objects to see their actual rigidity. An important note is that the evaluation data set only showed the points of spattering defects in an object, there may be other types of defects as well.

The evaluation set created using the aforementioned methodology is available for two objects referred to here as objects A and B. For the remaining objects, the total number of spatters across the objects was available. An example of located spatters for object B can be seen in Table 5.1. The first column 'Z' in Table 5.1 represents the layer number, the next two columns 'X' and 'Y' represent the location of the spatter in two dimensions, and the last column 'Object' represents the Object number. The objects used for evaluation in this thesis were object A which was printed with improper parameters, object C which was printed with optimal parameters and object B which had parameters somewhere in between.

5.3 Metrics of Interest

As explained in the problem description, besides detection success, the other main metrics are *latency*, i.e. time from the input of image to the output of clusters and geometric summary of those clusters, and *throughput* i.e. how many images can be handled per time unit. For the L-PBF printer used in this thesis, there is a threshold of 3 seconds per layer before the new layer starts being built and the goal was to have the analysis done before that. For plotting the latency a boxplot is used with the interquartile range (IQR) method [34]. IQR is simply the distance between the first quartile (Q1) and the third (Q3). The box itself spans from the 25th percentile to the 75th and the whiskers from $1.5 * IQR$ below Q1 to $1.5 * IQR$ above Q3.

For the classification, the main metrics considered were the ones mentioned in Section 2 for binary classification: *Accuracy*, *Precision*, and *Recall*. Another metric is the *number of outliers belonging to an intra-layer cluster*, which represents the percentage of outlier points that become part of intra-layer clusters. As discussed

Z	X	Y	Object
297	948	1073	B10
297	949	1073	B10
306	928	1047	B10
307	958	1032	B10
307	959	1032	B10
307	959	1033	B10
309	968	1053	B10
310	935	1035	B10
310	936	1035	B10
314	943	1065	B10
314	943	1066	B10
314	944	1066	B10
314	944	1067	B10
315	939	1062	B10
318	956	1069	B10
327	946	1059	B10
327	946	1060	B10
328	923	1049	B10
328	924	1049	B10
328	923	1050	B10
328	924	1050	B10
332	944	1066	B10
332	944	1067	B10
332	945	1067	B10

Table 5.1: Evaluation dataset locating spatters for object B

in Section 4.4 if this number is low it is a sign of the outliers being randomly distributed in the layer. Another metric studied was the number of true positives that end up in an inter-layer cluster which gave an idea of how prevalent it is for defects to span across layers.

5.4 Evaluation Setup

The method was evaluated using the OT images but, as mentioned in Section 4.1 each object needed to be assessed individually.

To achieve this, the region of interest of each object from the OT image was cropped. The precise cropping of each object could be accomplished through two main methods. The first method involved using layout and precise positional dimensions of objects on the build plate, which can be converted into positional dimensions of objects on the image. The second method is to utilize a tool that facilitates manually locating the position dimensions of each object individually, though these dimensions may not be as precise. Using this information, each object was cropped from the frames captured by the monitoring camera. In this thesis, the sides of the objects in the images were located manually by looking at the pixel position at the start and end of the objects. These cropped images of each object from each layer were written to the local file system, with the file naming convention including the layer number for easy identification and retrieval. As mentioned in Section 4.3 our approach has the ability to look at both cold and hot outliers but as the evaluation data set only focused on warm thermal anomalies this work did the same.

The file monitoring operator is used to read images from the disk and send the images into the data stream ordered by their layer number.

For the inter-layer operator, the window size was set to unbounded to not miss any overlapping clusters and it was sent to the sink, the standard output, after each layer had gone through the inter-layer clustering operator.

The following results were found when running the implementation on a Lenovo Thinkpad T480s with 16 GB of RAM and running Flink version 1.19.0 on Windows 11. The latency test was run with no other applications running except the OS applications.

5.5 Input Parameters

The value of different input parameters would affect what the results would look like. The input parameters for the algorithms used were:

- **Neighborhood size, η:** The size of a neighborhood is used to calculate the average Z-score for a point to find outliers.
- **Outlier percent, ξ:** The percent of points that are considered as outliers.
- **Epsilon, ϵ:** The distance in which there must be at least $minPts$ for a point to be considered a core point according to DBSCAN.
- **minPts:** The number of points that must be in the distance ϵ of a point to qualify it as a core point in DBSCAN.

For the input parameters of DBSCAN, we decided to set ϵ as 1.5 pixels and $minPts$ as 7. This was decided by a desire to keep the clusters as compact as possible to not accidentally overlap different defects and to ignore noise. The other two parameters, ξ, and η, were varied for different results.

5.6 Results

5.6.1 Classification

In this section, the performance of the proposed method is evaluated in terms of accuracy, precision, and recall against the results in the evaluation data set.

The evaluation aimed to assess how often a hit from the evaluation data set is found in the clusters of **one** layer, i.e. intra-layer clusters, and how many hits from the evaluation dataset are found in cross-layer clusters, i.e. inter-layer clusters. Furthermore, the impact of various input parameters on the output was examined. For the classification performed, the following definitions were applied to the evaluations mentioned in Section 2.

- **True Positive (TP):** Number of clusters that contained spatter points.
- **False Positive (FP):** Number of clusters that did **not** contain a spatter point.

- **True Negative (TN):** Number of points that are correctly identified as not being spatter points and not belonging to any cluster. (No such points were found and therefore this number is always zero in the calculations for the reported results in this setup.)
- **False Negative (FN):** Number of spatter points that did not belong to a cluster.

In Figure 5.4 there are confusion matrices when running the method on objects A and B. Recall that object A was printed with the worst parameters, object B better, and object C with optimal ones. This confusion matrix was found using ξ as 5% - 15%, a neighborhood size of 10, ϵ as 1.5, and $minPts$ as 7. The DBSCAN parameters were set strictly to minimize the chance of two defects becoming one cluster. The true negatives in these images can be disregarded as there were no sensical numbers to put here in this context.

In Table 5.2 the evaluation metrics of object A given the numbers in Figure 5.4 (a), 5.4 (b), 5.4 (c) are presented.

Percent of outliers (ξ)	Accuracy	Precision	Recall	TPs in Inc. clusters*
5%	38%	38.7%	95.5%	53.7%
10%	21.6%	21.7%	98.2%	75.9%
15%	14.8%	14.8%	98.3%	85.5%

Table 5.2: Table of the different metrics found of object A based on the percentage of outliers given. *Number of true positives that belonged to an inter-layer cluster.

What these metrics show is that almost all hits in the evaluation set were covered by a cluster (high recall) but there were also many clusters that did not have a hit (low precision). As mentioned earlier in Section 5.2.2 the baseline used for evaluation only looked for spattering defects and there may be other types of defects as well. This means that a false positive may still be a defect. Also, the evaluation data set might contain false positives as well.

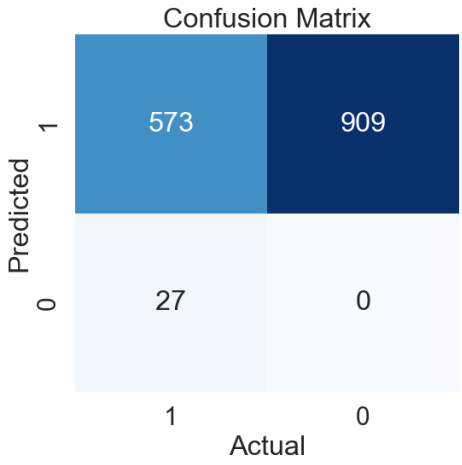
The results also showed that a majority of the true positive clusters were part of a cross-layer cluster as well, indicating that the defects span across layers.

For object B we got the evaluation metrics in Table 5.3.

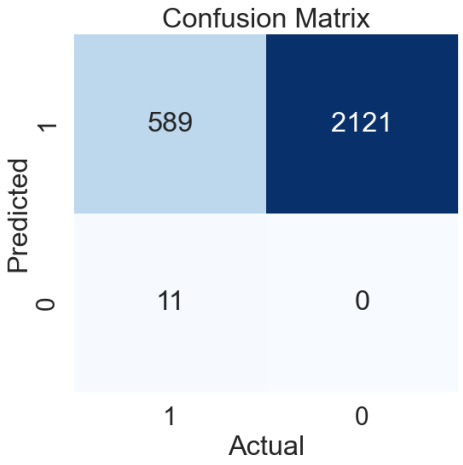
Percent of outliers (ξ)	Accuracy	Precision	Recall	TPs in Inc. clusters*
5%	25.7%	26%	96.4%	46.3%
10%	13%	13%	99%	61.2%
15%	7.9%	8%	99%	86.7%

Table 5.3: Table of the different metrics found of object B based on the percentage of outliers given. *Number of true positives that belonged to an inter-layer cluster.

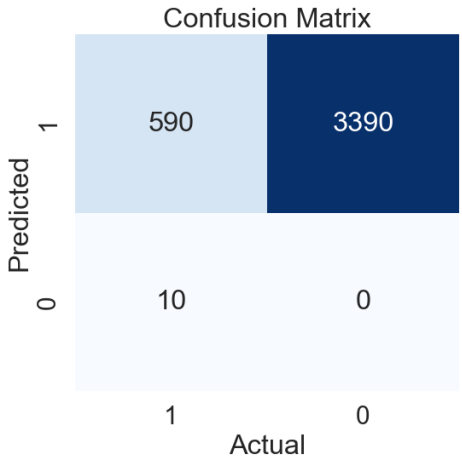
For this object, recall remains very high but the accuracy and precision get lower. This might mean that while the spattering defects decrease thanks to better parameters, the number of other types of defect might stay the same, thus creating more



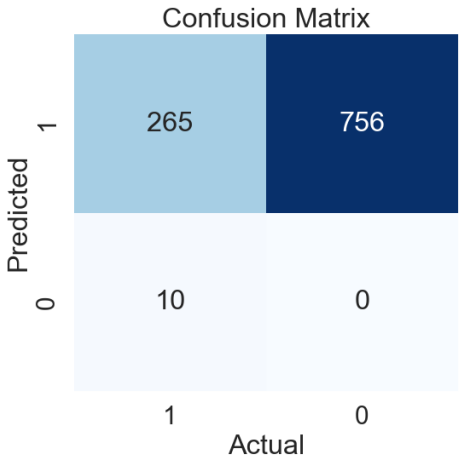
(a) Object A with ξ set to 5%.



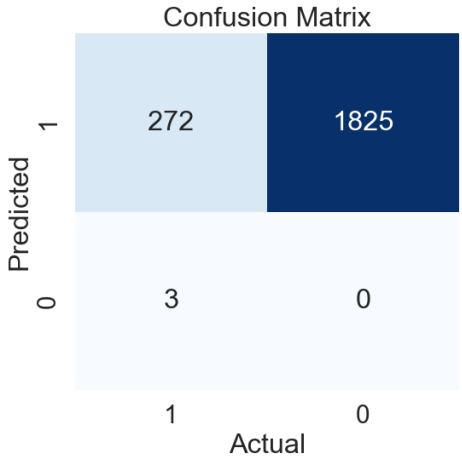
(b) Object A with ξ set to 10%.



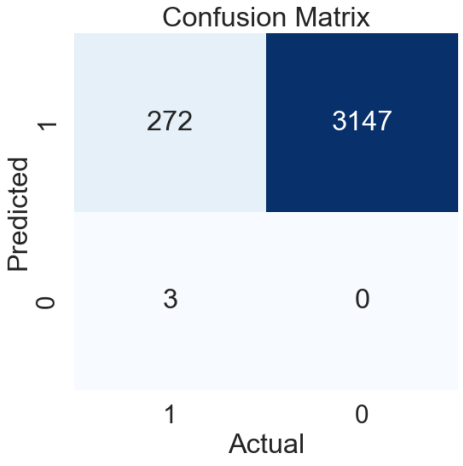
(c) Object A with ξ set to 15%.



(d) Object B with ξ set to 5%.



(e) Object B with ξ set to 10%.



(f) Object B with ξ set to 15%.

Figure 5.4: Confusion matrices for objects A and B with different ξ values: (a) A with 5%, (b) A with 10%, (c) A with 15%, (d) B with 5%, (e) B with 10%, (f) B with 15%.

false positives than true positives. As the number of outliers increases the number of incremental clusters increases as well, this is reasonable since more outliers can give more and larger clusters increasing the chance of an overlap.

For object C classification was not possible as there were no hits in the evaluation data set. Instead, we looked at the transition rate of outliers that is visible in Figure 5.5. In the figure, it is clear that object B had a constant lower amount of outliers that belonged to clusters compared to object A, and C had a constant lower amount compared to object B. This shows that the more defective objects had more outliers clustered together while the higher quality objects had more isolated outliers.

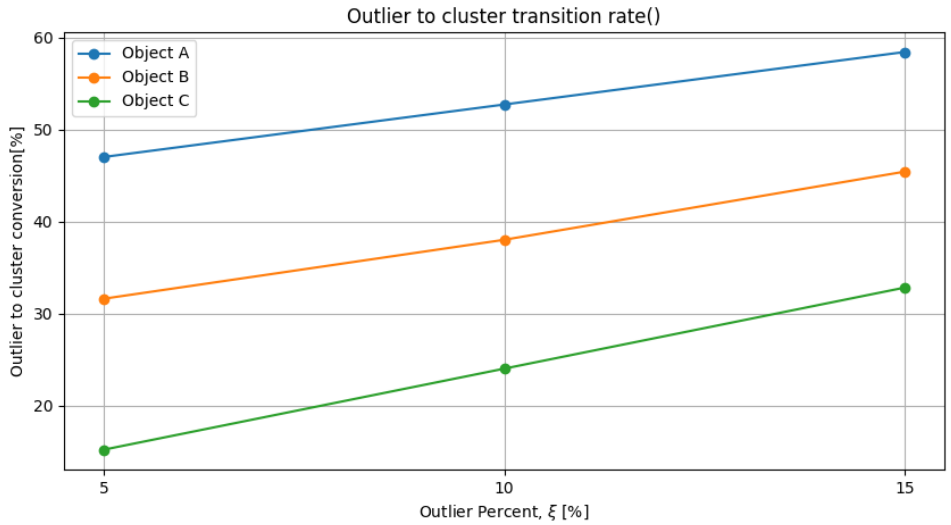


Figure 5.5: The average percent of outliers in objects A, B, and C that belonged to an intra-layer cluster, i.e. how many outliers ended up being part of a cluster

.

Another input parameter for the outlier detection was neighborhood size, previously set to 10 in the classification shown above. While there were some performance improvements with increased neighborhood size it was quite small, at least for recall. In Figure 5.6 and 5.7 there are plots for the change in accuracy, precision, and recall with variable neighborhood size. As visible the accuracy and precision got lower after neighborhood size 5 but this could be due to finding other defects in larger neighborhoods that are not present in the evaluation data set.

5.6.2 Geometric Summaries

For the different objects, the volumes of the geometric summaries looked as presented in Tables 5.4, A.1 and A.2. While the max and min varied as when for example object B's max is larger than object A's for an outlier percentage of 15%, the mean showed clearly that the better objects had smaller incremental clusters.

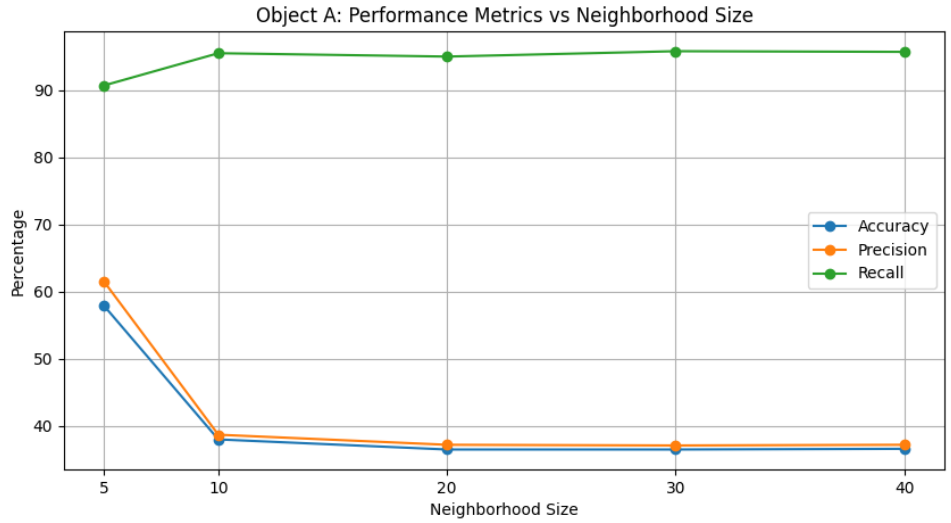


Figure 5.6: The resulting evaluation metrics of object A with different neighborhood sizes, with ξ set to 5%.

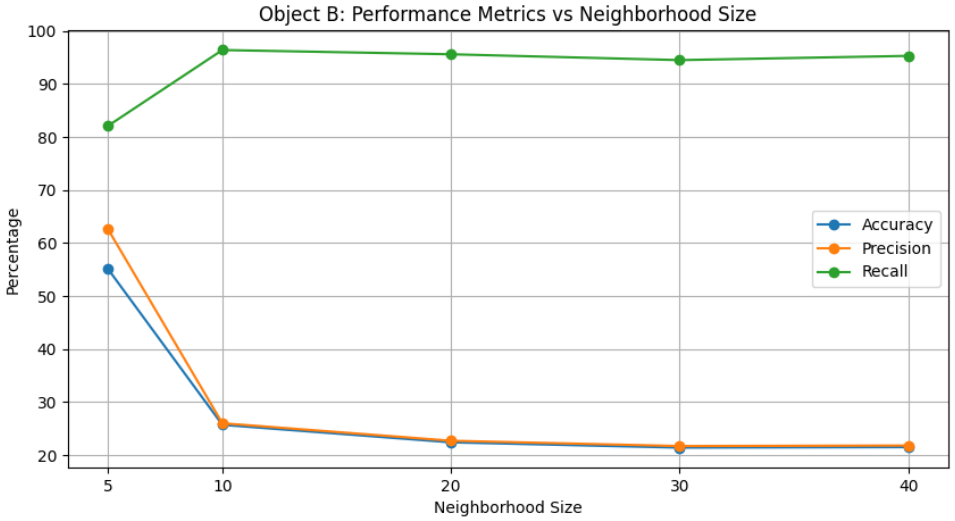


Figure 5.7: The resulting evaluation metrics of object B with different neighborhood sizes, with ξ set to 5%.

	Maximum	Mean	Minimum
Object A	660	98.1	12.3
Object B	400	74.8	15.8
Object C	140.3	45.6	12.6

Table 5.4: The maximum, minimum, and mean volume of the geometric summaries given in the number of pixels. Found when ξ was 5%.

5.6.3 Timeliness

The latency was calculated by noting the time at the input of an image to the outlier detection operator to the output of the clustering/geometric summary operator. This was run 10 times on all 250 images in the data set to get an average latency for each image. In Figure 5.11 the latency for processing one image as a function of different outlier percentages can be seen. In Figure 5.12 there is a graph showing the latency of the outlier detection operator as a function of the neighborhood size. According to the complexity analysis, as the image size is constant, this should scale linearly with the neighborhood size. This is also the case as the mean is around the double for each box.

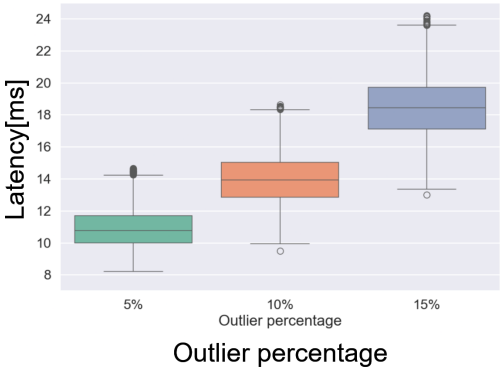


Figure 5.8: Object A Latency

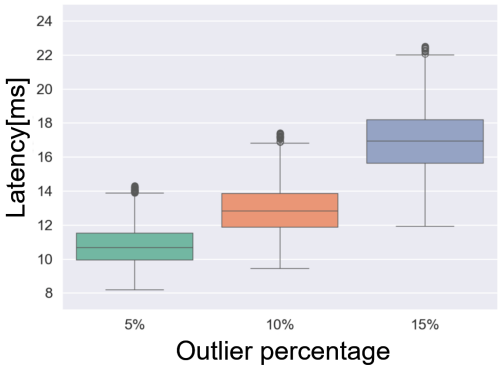


Figure 5.9: Object B Latency

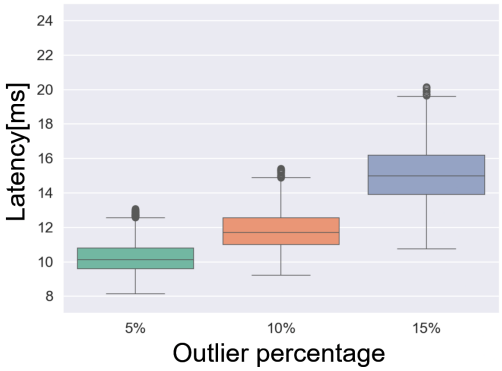


Figure 5.10: Object C Latency

Figure 5.11: The latency of the stream pipeline using different outlier percentages. The boxplots span from the 25th to the 75th percentile, and the whiskers were calculated according to the IQR method.

	5%	10%	15%
Object A	91.5	71.4	54.1
Object B	92.5	77.2	58.8
Object C	97.3	84.4	66.1

Table 5.5: Throughput of each object with different outlier percentages. The unit is images/second.

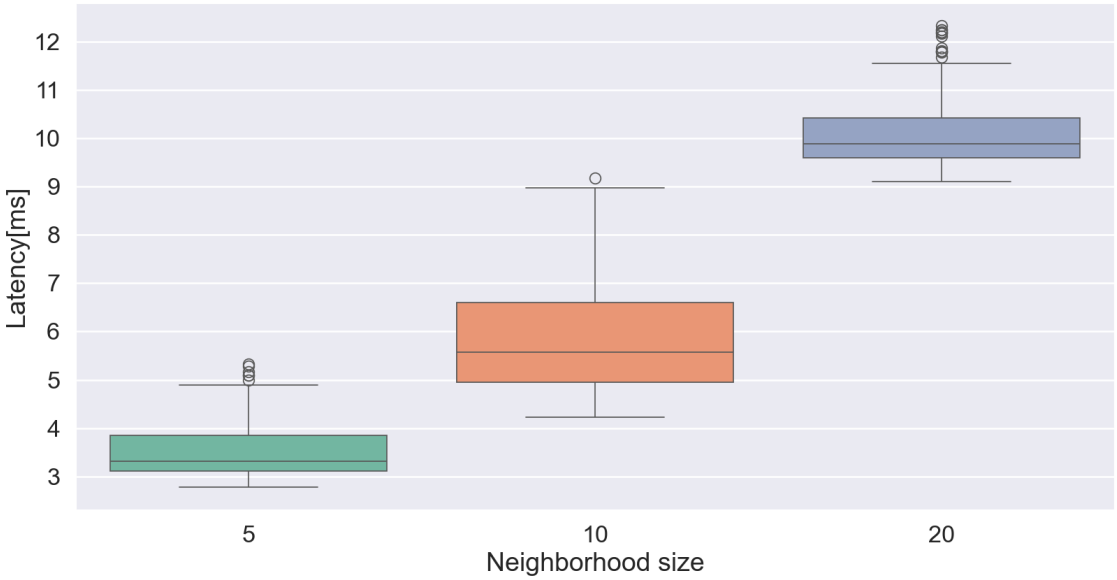


Figure 5.12: How the latency of the outlier operator varies with different neighborhood sizes.

The throughput can be seen in Table 5.5. For low percentages, it was difficult to see differences in throughput likely due to the data being very small. At high percentages, however, we saw larger differences and the output followed logically that the more defect-infested objects had lower throughput.

5.7 Discussion

The results of this thesis show that clustering of thermal outliers can with good accuracy find at least certain types of defect. The very high recall even for a lower amount of outliers shows that the methodology has a correlation to the physical properties of the object. The low conversion rate of outliers to clusters in object C signifies that the outliers are more randomly scattered in object C than in object A. Taking this with the fact that it has been proven that object C has fewer defects than object A it seems likely that the clusters, or at least a subset of them, represent true defects.

The low precision found in the classification could be due to a lack of data in the evaluation data set. As it only looked at spattering defects there might be other types of defects found by the clusters but not in the evaluation data set resulting in a false positive evaluation. What we know for a fact is that object C is more rigid than object B and object A and the low conversion rate of outliers to clusters points to that as well.

The geometric summaries also show a significant decrease in mean size when run on the more rigid objects, object B and C, instead of object A. This shows that the incremental clusters are smaller which most likely stems from them having a lower outlier-to-cluster conversion rate and thereby a smaller chance of overlapping.

As for the timeliness, it showed low latency and the ability to process images well below the QoS threshold of 3 seconds. Even accounting for the worst latency and processing all objects on the build plate sequentially would only yield about $25ms * 64 = 1.6s$ of latency meaning significantly faster than 3 seconds.

The complexity of the algorithm is relatively low especially when considering that most clusters don't overlap and can be checked in constant time. But since the clusters are so small, only 20-30 points each, the calculations are so fast that there may not be a need to adjust the algorithm for these types of datasets. Instead focus could be on using high-performance computers if a decrease in latency is needed. As the printers themself are a large investment, adding on to that the price of a high-performance computer should not be a problem. The complexity would however be more important when building larger objects that yield larger defects as that would yield larger clusters meaning more computational need.

Compared to many of the results that have been previously found this method uses unsupervised machine learning, which leads the way to work on AM defect detection without the need for labeled data and for working with a general data set of images. Compared to the work done in [11] this solution showed a lower latency at a max of about 25ms while the average in [11] was about 48 ms.

5.8 Sources of Error

One potential source of error when comparing this methodology to using live data is that the images used contain the maximum radiance of layers. This would not be the case with images streamed from the printer live as the temperature would fluctuate when the laser travels across the bed. This could of course be solved by either averaging the incoming images and running the implementation on one image for each layer or finding the max output for each pixel.

The input parameters are another source of error as too large input parameters can create clusters that do not accurately pinpoint defects in the layer but rather increase in size and make the location of the defect more diffuse. A too large outlier percentage would also give a higher probability of a cluster overlapping with a defect without accurately describing the area of the defect itself.

6

Conclusions and Future Work

This thesis showed that spatial outlier detection along with clustering of the points found can identify defects in materials caused by spattering and showed that it is viable to detect other types of defects as well. The method introduced gave a clear location and geometry of suspicious areas in one layer and also across layers. It was able to quantify the volumes of suspected defects in a timely manner on par with existing research today.

For future work, there are several different aspects of this thesis to look at:

- A test of this methodology could be done on X-ray data as ground truth. That could give a more accurate view of the correlation of a cluster and a defect as the ground truth in this work has not looked at all types of defects. This would also show how accurately the clusters that span across layers track defects.
- A method to determine whether an object should be treated as usable or not working on live data based on the parameters given in this result. For example, specific values for outlier-to-cluster conversion could indicate whether an object contains a lot of flaws and so could large AABBs of its clusters.
- As discussed before, in Section 5.7, the complexity should not be a problem for data sets of this size, but if there should be a need for latency reduction in future research there could be an investigation into using parallel versions of DBSCAN or improved hardware such as GPUs.
- Another extension of this work would be to look at cold outliers and not only hot outliers as has been done here, due to the evaluation set only looking at hot spots. This could make it possible to detect other types of defects such as lack-of-fusion pores but would of course require a different evaluation set.
- Continuing work on this methodology and building a framework that can not only detect defects in layers but predict when they will spread across layers would be a useful addition to the field. This could perhaps be done by looking at a mix of parameters from this work such as outlier-to-cluster transition rate and size of outliers and geometric summaries.
- On a related note trying to fix defects in previous layers by looking at their attributes from this work would also lead the field closer to a closed-loop solution of printing in L-PBF.

- Switching from AABBs to a geometric summary method that produces a tighter fit for arbitrarily shaped data points would also be an interesting future work. Doing so can give a more detailed view of the clusters even though it would probably yield a similar change of size as found in this result.
- Since the images are presented as a matrix they can easily be divided into sub-jobs in order to process multiple objects of one print in parallel. This thesis did not do a dive into how much parallelism can improve the result and in what aspects it can be used. That could be an interesting future work.

Bibliography

[1] P. Lu, Z. Cheng-Lin, L. Tong, L. Xin-Yu, L. Jiang-Lin, L. Shun, W. Wen-Hao, and Z. Heng-Hua, "Molten pool structure and temperature flow behavior of green-laser powder bed fusion pure copper," *Materials Research Express*, vol. 9, no. 1, p. 016 504, Jan. 2022, ISSN: 2053-1591. DOI: 10.1088/2053-1591/ac327a.

[2] S. Cao, H. Wang, X. Lu, J. Tong, and Z. Sheng, "Topology optimization considering porosity defects in metal additive manufacturing," *Applied Sciences*, vol. 11, no. 12, p. 5578, Jun. 2021, ISSN: 2076-3417. DOI: 10.3390/app11125578.

[3] H. Zhang, C. K. P. Vallabh, and X. Zhao, "Influence of spattering on in-process layer surface roughness during laser powder bed fusion," *Journal of Manufacturing Processes*, vol. 104, pp. 289–306, Oct. 2023, ISSN: 1526-6125. DOI: 10.1016/j.jmapro.2023.08.058. [Online]. Available: http://dx.doi.org/10.1016/j.jmapro.2023.08.058.

[4] G. Budzik, J. Woniak, A. Paszkiewicz,. Przeszowski, T. Dziubek, and M. Dbski, "Methodology for the quality control process of additive manufacturing products made of polymer materials," *Materials*, vol. 14, no. 9, p. 2202, 2021.

[5] M. Grasso, A. Remani, A. Dickins, B. M. Colosimo, and R. K. Leach, "In-situ measurement and monitoring methods for metal powder bed fusion: An updated review," *Measurement Science and Technology*, vol. 32, no. 11, p. 112 001, Jul. 2021, ISSN: 1361-6501. DOI: 10.1088/1361-6501/ac0b6b.

[6] N. Panahi, "Multi-purpose parameter development for high productivity in laser powder bed fusion of in718," M.S. thesis, Chalmers University of Technology, 2020.

[7] G. Mohr, S. J. Altenburg, A. Ulbricht, P. Heinrich, D. Baum, C. Maierhofer, and K. Hilgenberg, "In-situ defect detection in laser powder bed fusion by using thermography and optical tomographycomparison to computed tomography," *Metals*, vol. 10, no. 1, p. 103, Jan. 2020, ISSN: 2075-4701. DOI: 10.3390/met10010103.

[8] S. Moylan, E. Whitenton, B. Lane, and J. Slotwinski, "Infrared thermography for laser-based powder bed fusion additive manufacturing processes," in *AIP Conference Proceedings*, AIP Publishing LLC, 2014. DOI: 10.1063/1.4864956. [Online]. Available: http://dx.doi.org/10.1063/1.4864956.

[9] H. C. M. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, Feb. 2014, ISBN: 9781139058940. DOI: 10.1017/cbo9781139058940.

[10] M. Srivastava, S. Rathee, V. Patel, A. Kumar, and P. G. Koppad, "A review of various materials for additive manufacturing: Recent trends and processing issues," *Journal of Materials Research and Technology*, vol. 21, pp. 2612–2641, 2022, ISSN: 2238-7854. DOI: https://doi.org/10.1016/j.jmrt.2022.10.015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2238785422015526.

[11] E. Sievers, "Data analysis for defect monitoring in additive manufacturing– applying machine learning to predict porosity in l-pbf," M.S. thesis, Chalmers University of Technology, 2023.

[12] N. Nudelis and P. Mayr, "A novel classification method for pores in laser powder bed fusion," *Metals*, vol. 11, no. 12, p. 1912, 2021.

[13] M. Dharnidharka, U. Chadha, L. M. Dasari, A. Paliwal, Y. Surya, and S. K. Selvaraj, "Optical tomography in additive manufacturing: A review, processes, open problems, and new opportunities," *The European Physical Journal Plus*, vol. 136, no. 11, p. 1133, 2021.

[14] V. Gulisano, M. Papatriantafilou, Z. Chen, E. Hryha, and L. Nyborg, "Towards data-driven additive manufacturing processes," in *Proceedings of the 23rd International Middleware Conference Industrial Track*, ser. Middleware 22, ACM, Nov. 2022. DOI: 10.1145/3564695.3564778.

[15] *Apache flink documentation*, 2024. [Online]. Available: https://nightlies.apache.org/flink/flink-docs-stable/ (visited on 06/03/2024).

[16] *Apache flink documentation: Architecture*, 2024. [Online]. Available: https://nightlies.apache.org/flink/flink-docs-release-1.19/docs/concepts/flink-architecture/ (visited on 05/26/2024).

[17] *Yarn — classic.yarnpkg.com*, https://classic.yarnpkg.com/lang/en/docs/, [Accessed 20-07-2024].

[18] *Apache Mesos — mesos.apache.org*, https://mesos.apache.org/documentation/latest/, [Accessed 20-07-2024].

[19] *Kubernetes Documentation — kubernetes.io*, https://kubernetes.io/docs/home/, [Accessed 20-07-2024].

[20] UnskewData, *Stream Processing with Apache Flink. Part-1 — contactunskewdata*, https://medium.com/@contactunskewdata/stream-processing-with-apache-flink-part-1-8dd959427f5a, [Accessed 05-06-2024], 2024.

[21] F. Hueske and V. Kalavri, *Stream processing with Apache flink*. Sebastopol, CA: O'Reilly Media, Apr. 2019.

[22] A. Boukerche, L. Zheng, and O. Alfandi, "Outlier detection: Methods, models, and classification," *ACM Computing Surveys*, vol. 53, no. 3, pp. 1–37, Jun. 2020, ISSN: 1557-7341. DOI: 10.1145/3381028.

[23] A. Lindholm, N. Wahlström, F. Lindsten, and T. B. Schön, *Machine learning: a first course for engineers and scientists*. Cambridge University Press, 2022.

[24] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *kdd*, vol. 96, 1996, pp. 226–231.

[25] NSHipster, *Dbscan illustration*, Accessed: 2024-07-01, n.d. [Online]. Available: https://github.com/NSHipster/DBSCAN.

[26] H. Najdataei, Y. Nikolakopoulos, V. Gulisano, and M. Papatriantafilou, "Continuous and parallel lidar point-cloud clustering," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, IEEE, Jul. 2018. DOI: 10.1109/icdcs.2018.00071.

[27] A. Keramatian, V. Gulisano, M. Papatriantafilou, and P. Tsigas, "Mad-c: Multi-stage approximate distributed cluster-combining for obstacle detection and localization," *Journal of Parallel and Distributed Computing*, vol. 147, pp. 248–267, Jan. 2021, ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2020.08.013.

[28] A. Keramatian, V. Gulisano, M. Papatriantafilou, and P. Tsigas, "Parma-cc: A family of parallel multiphase approximate cluster combining algorithms," *Journal of Parallel and Distributed Computing*, vol. 177, pp. 68–88, Jul. 2023, ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2023.02.001.

[29] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire, "Intersection test methods," in *Real-Time Rendering*. A K Peters/CRC Press, Aug. 2018, pp. 941–991, ISBN: 9781315212593. DOI: 10.1201/b22086-22.

[30] S. M. Estalaki, C. S. Lough, R. G. Landers, E. C. Kinzel, and T. Luo, "Predicting defects in laser powder bed fusion using in-situ thermal imaging data and machine learning," *Additive Manufacturing*, vol. 58, p. 103 008, Oct. 2022, ISSN: 2214-8604. DOI: 10.1016/j.addma.2022.103008.

[31] T. H. Cormen and C. E. Leiserson, *Introduction to Algorithms, fourth edition*, en. London, England: MIT Press, Apr. 2022.

[32] *Powder metallurgy and additive manufacturing — chalmers.se*, https://www.chalmers.se/en/departments/ims/research/materials-and-manufacture/powder-metallurgy-and-additive-manufacturing/.

[33] C. Schwerz, A. Raza, X. Lei, L. Nyborg, E. Hryha, and H. Wirdelius, "In-situ detection of redeposited spatter and its influence on the formation of internal flaws in laser powder bed fusion," *Additive Manufacturing*, vol. 47, p. 102 370, Nov. 2021, ISSN: 2214-8604. DOI: 10.1016/j.addma.2021.102370. [Online]. Available: http://dx.doi.org/10.1016/j.addma.2021.102370.

[34] F. M. Dekking, C. Kraaikamp, H. P. Lopuhaä, and L. E. Meester, *A Modern Introduction to Probability and Statistics: Understanding why and how*. Springer Science & Business Media, 2006.

A

Appendix 1

	Maximum	Mean	Minimum
Object A	1235	134.7	12.3
Object B	624	93.4	12.3
Object C	273	68.3	12.3

Table A.1: The maximum, minimum, and mean volume of the geometric summaries given in the number of pixels. Found when ξ was 10%.

	Maximum	Mean	Minimum
Object A	1848	206.7.8	12.3
Object B	2016	132.5	12.3
Object C	1306.3	102.9	12.3

Table A.2: The maximum, minimum, and mean volume of the geometric summaries given in the number of pixels. Found when ξ was 15%.

The code of the projet can be found on the guthub here: `https://github.com/Aeroplane007/MT-AM`