Detecting elementary particles
with Timepix3 detector

Department of Software and Computer Science Education

Supervisor of the master thesis: RNDr. František Mráz, CSc.
Study programme: Computer Science (N1801)
Study branch: IUI (1801T036)

Prague 2019
I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ........ date ............ signature of the author
I would like to thank my thesis advisor RNDr. František Mráz, CSc. of the Faculty of Mathematics and Physics, Charles University for his support and advises during the writing of the thesis.

I want to express my gratitude to the Institute of Experimental and Applied Physics, Czech Technical University in Prague and its director doc. Ing. Ivan Štekl, CSc. for long-term support.

I also would like to thank my advisor Benedikt Bergmann, MSc. for his guidance in the field of particle physics and continuous assistance with research. Additionally I am grateful for letting me participate and contribute in many physical experiments associated with this thesis, such as project Research infrastructure for experiments at CERN.

I appreciate the support of other members of the Institute of Experimental and Applied Physics, namely: Ing. Stanislav Pospíšil, DrSc., prof. RNDr. Michal Suk and Ing. Petr Burian, Ph.D.

I would also like to thank my friend Bc. Lukáš Jelínek for his expertise and help during the research and studies.
Title: Detecting elementary particles with Timepix3 detector

Author: Lukáš Meduna

Department: Department of Software and Computer Science Education

Supervisor: RNDr. František Mráz, CSc., Department of Software and Computer Science Education

Abstract: Timepix3 is capable of simultaneous measurement of energy and trajectory of ionizing particle with precision up to 1.5625 ns. Advanced data-driven mode and the Katherine readout present new challenges in data acquisition and analysis.

To process and separate events in the high flux of pixels, new framework was developed. It operates both in offline and online mode with high performance and can visualize and store processed data simultaneously.

The analysis of measured events is performed. Characterization and morphological analysis of resulting data are presented. Using the means of machine learning, an isolation of exotic events is done by classifying and filtering the unimportant data.

Results of the work show advantages of Timepix3 in combination with real-time processing and visualization. The results of classification are promising and the possibility of applying these models to the real-time measurement is investigated.

Keywords: Timepix3 Experimental physics Katherine readout Medipix
Contents

Introduction 4

1 Hybrid pixel detectors 5
  1.1 Principles ................................................. 5
  1.2 Medipix collaboration ........................................ 6
  1.3 Timepix3 .................................................. 6
  1.4 Readouts and data acquisition .............................. 8
  1.5 Calibration and corrections ................................ 9
  1.6 Source of data .............................................. 11

2 Data processing and visualization 13
  2.1 Architecture ............................................... 14
    2.1.1 Internal modules ....................................... 16
    2.1.2 Extensions .............................................. 16
    2.1.3 Lifecycle of components ................................. 17
    2.1.4 Configuration files ..................................... 18
  2.2 Input data sources ........................................ 19
    2.2.1 Katherine UDP communication .......................... 20
  2.3 Katherine file format ..................................... 22
  2.4 Clustering .................................................. 22
    2.4.1 Quadtree ................................................. 26
  2.5 Monitoring and storing results ............................. 28
    2.5.1 Filtering plugins ....................................... 29
    2.5.2 Event statistics overview ............................... 31
    2.5.3 Cluster integration window ....................... 33
    2.5.4 Saving the clusters .................................... 36
    2.5.5 ROOT files ............................................. 37

3 Timepix3 cluster data 39
  3.1 Cluster viewer and analysis software ..................... 39
    3.1.1 CSV file reader and data browser .................. 40
    3.1.2 ToA and ToT visualization ............................ 41
    3.1.3 3D visualization ...................................... 41
    3.1.4 Feature testing and manual labeling ................. 42
  3.2 Classification goals ...................................... 42
    3.2.1 Basic categories of clusters ........................ 42
    3.2.2 Beyond defined groups ................................ 44
  3.3 Feature engineering ....................................... 45
    3.3.1 Feature selection ...................................... 45
    3.3.2 Morphology of clusters ................................ 49
    3.3.3 Skeletonization ......................................... 51
Introduction

In the current particle physics research, visualization and understanding of particles and phenomena of the atomic and subatomic world is carried out in multiple ways. From huge detectors searching for undiscovered particles at accelerators to miniaturized devices in space measuring cosmic rays. One of such devices is the hybrid pixel detector Timepix3 [48]. It evolved from previous chips in the Medipix family [16].

Figure 1: Timepix at the International Space Station (In the right picture inside the yellow circle). On the left is a detail of Timepix sensor with USB interface [27]. The Timepix chip was sent to ISS by members of the Medipix collaboration at the Institute of Experimental and Applied Physics in Prague and the University of Houston, Texas. Its task is to monitor the radiation environment for ionizing dose assessment inside the space station [26].

Novel features of the Timepix3 are the simultaneous measurement of time (time binning 1.5625 $\mu$s) and energy in each pixel and the data-driven readout. These novelties require modification or redesign of existing software packages. New methodology for data analysis is needed to fully exploit the capabilities.

This thesis will guide us from the electronics through the data acquisition to the field of machine learning and data analysis. It is shown how particle interactions form tracks in the sensor matrix, which are reconstructed, visualized, analyzed and grouped according to characteristic features. These features indicate particle type and energy, necessary for the measurement of radiation field composition.

Goals of the thesis

The goal of the thesis is to develop framework for preprocessing of data from Timepix3 in real-time. This framework will visualize and store the processed data in convenient format. The second task is to identify and test on real data suitable methods for classification of measured events.

Outline of the thesis

We will begin with a simplified description of the physics behind the hybrid pixel detectors in Chapter 1. Specifically we will cover in more detail Timepix3 and
its capabilities used in the work. Crucial to our work is also the next electronic device in the measurement chain – the Katherine readout. At the edge of this device, our software work is beginning.

In Chapter 2, we will show the communication and control of the Katherine readout, how to get the raw data from the sensor and which formats are used. After getting the data in the form of a not-necessary time sorted stream of pixels (containing not only $x$ and $y$ coordinates, but also time of arrival of the pixel), we will describe in detail how we convert this stream into separated events that happened within microseconds on the Timepix3 chip. These data can then be used for visualization of particle events, statistics of the measurement or stored for further analysis. An example of one of the possible output formats is shown in the following figure, where we can observe all the events that took place in one second of measurement.

![Figure 2: Example of particle tracks observed within a one second integration window. It is similar to a picture from standard CMOS camera, but with a resolution of 256 × 256 pixels. The color depicts the energy deposited in a pixel. Details are explained later in the work.](image)

We will present a tool for browsing individual events in Chapter 4. Depending on the radiation field, Timepix3 can produce a huge amount of data. Not always all tracks are useful for further analysis. Thus, a real-time filtering and event categorization was added to the data acquisition chain. Saving only events of interest reduces disk space requirements significantly. To achieve that we will try to characterize individual tracks.

Finally, in Chapter 4, events with selected properties can be then classified into predefined categories. We will show the methods that we used to perform the classification and outline the directions of further research that can be build upon our work.
1. Hybrid pixel detectors

Ionizing particle detection plays an important role in multiple fields of science. Ranging from particle physics, radiation dosimetry and education to applied use in medical imaging. One of the milestones of the pursuit to capture and visualize the trajectories of ionizing particles was the development of the Wilson cloud chambers \[6\]. Cloud chambers played a role in the discoveries of the positron and muon, both awarded with the Nobel Prize. An example of a cloud chamber measurement is depicted in Figure 1.1.

Figure 1.1: Cloud chamber picture of tracks of ionizing radiation \[6\].

Due to improvements in semiconductor development and readout electronics such tracks can now be visualized in much smaller volumes, for example with the hybrid pixel detectors which are described in the following section.

1.1 Principles

Figure 1.2: Annotated model of a hybrid pixel detector \[47\].
The Timepix3 assembly is constructed of two main layers. These are the sensitive layer and the readout ASIC which are connected through so-called bump-bonds. The readout chip is connected to the motherboard using wire-bonds.

The sensitive layer (detection medium) is where the particle leaves its trace. It is semi-conductive material, usually silicon, gallium arsenide or cadmium telluride (CdTe). The thickness ranges from 300 µm to 1 mm for silicon or 1 mm up to 5 mm for CdTe.

The second part is an ASIC with a CMOS circuit representing individual pixels. The readout logic for individual pixels is placed here.

Particles passing through the detection medium elevate electrons from the valence band to the conduction band leaving a hole (missing electron, positive charge) in the valence band and an electron (negative charge) in the conduction band. Due to the applied configurable bias voltage between the top plate (common for all pixels) and individual pixels the charge carriers drift towards the electrodes at the bottom of sensor chip. The induced currents are shaped, amplified and digitized. As a result, Timepix3 sends information about energy deposition and time to the readout interface.

1.2 Medipix collaboration

Since 90’s there have been four active Medipix collaborations at CERN which focused on developing hybrid pixel detectors. Through the past years multiple devices were developed in these collaborations.

Medipix1, also known as Photon-counting-Chip was the first of the family. It used CMOS technology with resolution 64 × 64 pixels. It was capable of counting the number of hits per pixel in a given time window.

Medipix2 brought smaller pixel size of 55 µm and higher resolution of 256 × 256. However, it still did not supported additional modes of measurement other then counting mode.

Timepix is based on the Medipix2 device with the same dimensions, but with upgraded capabilities. The main improvements were the addition of two new measurement modes. The time of arrival is capable of recording when the pixel was hit and the time over threshold can measure how big was the charge.

Medipix3 has an improved front-end architecture and better pixel clarity of measurement due to the elimination of the charge sharing in the semiconductor detector. It still operates only in counting mode.

Timepix3 is currently the state-of-the art hybrid pixel technology. As we use this device in our thesis, we cover it more in depth in the next section.

1.3 Timepix3

Timepix3 was developed as the successor to the previous Timepix chip and further extended its capabilities. The chip was developed and funded by Medipix3 collaboration. It keeps its pixel size at 55 × 55 µm but improves the timing resolution. Particle tracking was seen as the main application. However, even

\footnote{Complementary metal-oxide-semiconductor – technology how the integrated circuit is constructed.}
pixel counting mode is still present with up to 40 \(Mhits/cm^2/s\). Because we will work solely with data from this chip in the rest of the thesis, we need to define more precisely some of the important technical terms. An overview of the chip capabilities is given in Table 1.2.

Table 1.1: Timepix3 general properties [48].

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMOS technology</td>
<td>130 nm</td>
</tr>
<tr>
<td>Pixel resolution</td>
<td>(256 \times 256)</td>
</tr>
<tr>
<td>Pixel size</td>
<td>(55 \times 55 \mu m)</td>
</tr>
<tr>
<td>Acquisition modes</td>
<td>Charge and time</td>
</tr>
<tr>
<td></td>
<td>Time only</td>
</tr>
<tr>
<td></td>
<td>Event counting and integral charge</td>
</tr>
<tr>
<td>Dead time per pixel</td>
<td>(475 \text{ ns} + \text{pulse time of time over threshold clock})</td>
</tr>
<tr>
<td>(time needed to reset the pixel after readout)</td>
<td></td>
</tr>
<tr>
<td>Time resolution</td>
<td>(1.5625 \text{ ns at 640 MHz})</td>
</tr>
<tr>
<td>Output bandwidth</td>
<td>Up to 5.12 Gbps</td>
</tr>
</tbody>
</table>

**Time of arrival**

Time of arrival (ToA) is the time in nanoseconds (ns) when the charge in the pixel surpass the given energetic threshold. The precision is given by clock frequency. The time of arrival of a cluster (i.e. set of pixels, defined later in the thesis) is the minimal ToA of all pixels in the cluster, unless stated otherwise.

**Time over threshold**

Time over threshold (ToT) is a positive integer number of clock counts. It measures the energy deposition in a pixel by sampling the time the signal in a pixel stays above a preset threshold. After the calibration process, the pulse length can be assigned to energy values.

The Timepix3 chip presents two readout modes. Firstly, already existent in Timepix, the frame-based readout mode. In this mode, after given time period the whole matrix of \(256 \times 256\) pixels is read out. Secondly, the data-driven mode was developed. In this mode, pixel information after hit (i.e. charge was over the threshold) is sent right away to the readout interface. After the dead-time of less than 1 \(\mu s\) pixel is ready to measure again. Thus each pixel asynchronously reports hits. This mode has two major advantages:

- The frame length has to be set manually, that causes problem in varying environments (e.g., ATLAS experiment where the flux fluctuate between beam collisions). Therefore we need to adjust the frame length so that we
do not over/under expose the frame. In contrast, the data-driven mode allows continuous measurement in changing radiation fields.

- In data-driven mode, events are separated and are not overlapping. In the frame-based mode, if two events happen in one frame and are overlapping then it is difficult, if not impossible, to separate the events.

- The data-driven mode is less demanding on bandwidth. Only hits are transferred, opposed to whole matrices in frame-based mode.

The downside is that when we want to recreate the events from pixels, we need to do a non-trivial procedure described in the following chapter. In this thesis, we use the charge and time acquisition mode combined with the data-driven readout mode. Simultaneous ToA and ToT measurement is enabled by the presence of multiple clocks. The schematics are shown in Figure 1.3.

![Figure 1.3: Schematics of a clock in the Timepix3 chip during ToA and ToT mode](48).

When the charge exceeds the given programmable threshold, the 640 MHz clock is started. The clock is stopped with the rise of 40 MHz clock. Let the number of ticks of the 640 MHz clock be called fine ToA. From this moment the time over threshold (ToT) measurement is started, also using the 40 MHz clock. In the same moment, the rough ToA is latched. Let the number of ticks from the measurement beginning (including the latched tick) be rough ToA. The ToT clock is ended when the charge falls below the threshold. To compute the final ToA in nanoseconds we use the formula:

$$ToA = \text{roughToA} \times 25 - \text{fineToA} \times 1.5625$$

where the constants are derived from the clock frequencies:

$$\frac{1}{640 \times 10^6} = 1.5625 \times 10^{-9}s, \quad \frac{1}{40 \times 10^6} = 25 \times 10^{-9}s.$$

1.4 Readouts and data acquisition

Timepix3 needs to be connected to an active readout device which communicates with the chipboard. Different readout devices were developed, such as AdvaDAQ
In the presented work, we used the in-house developed Katherine readout. Figure 1.4: Katherine readout device without Timepix3 chip. Backside (shown in figure) has RJ-45 Ethernet port, GPIO port and DC input. In front VHDCI (Very-High-Density-Cable-Interconnect) 68-pin connector for the Timepix3 is present and LEMO connector for bias.

The Katherine device was developed as a compact device with connection to a measurement computer via Ethernet cable while using standard UDP protocol. This allows the operation of the device in remote or unaccessible places (i.e. with high radiation), in contrast to devices using USB connection (the USB interface has limited working range). The connection between the Timepix3 chip and Katherine readout has been tested in four configurations:

Table 1.2: Connection options between Katherine readout and Timepix3 chip.

<table>
<thead>
<tr>
<th>Type of cable</th>
<th>Maximum hit rate</th>
<th>Distance m</th>
</tr>
</thead>
<tbody>
<tr>
<td>VHDCI F/M extending</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>VHDCI F/M extending</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Ethernet cat.7 (with extenders)</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>Ethernet cat.7 (with extenders)</td>
<td>5</td>
<td>100</td>
</tr>
</tbody>
</table>

In section 2.2 we will show the communication with the Katherine readout, its format and its data usage in the Clusterer software.

1.5 Calibration and corrections

As the response of each individual pixel in the Timepix3 matrix is never completely identical, calibration procedures have to be applied for each pixel separately.
Figure 1.5: Timepix3 device (in the center of figure) in the Large Hadron Collider in CERN. Due to high radiation, the Katherine readout electronics were placed over 100 m away from the chip in a cavern. Timepix3 measures the radiation field in the tunnel continuously.

The relation between energy and ToT count is known (see article [39]) and can thus be approximated. The process is known as **energy calibration**. We can obtain the necessary variables of the energy equation by irradiating the sensor with characteristic x-ray fluorescence line photons from different sources (indium, zirconium, copper, nickel, and titanium) and radioactive isotope of americium. Thus we can compute the approximation of the equation parameters. The approximated energy versus ToT can be seen in Figure 1.6.

![Figure 1.6: Approximated sensor response to the deposited energy using multiple sources with known energy [39].](image)

For the matrix of floating point parameters $a, b, c, t$ (each pixel has its own characteristics) we calculate *energy* in keV by:

$$ToT = a \ast energy + b - \frac{c}{energy - t}$$
Due to comparison with the threshold, the time of arrival has small error dependent on the energy deposited in the pixel as seen in Figure 1.7.

![Figure 1.7: Time of arrival inaccuracy. Although all events have the same peak time, the calculated ToA is different. This effect can occur when four neighboring pixels are hit at once](image)

Timewalk correction is introduced for each pixel with its threshold (in keV) to compensate for this error:

\[
ToA = ToA - \frac{a}{(\text{energy} - \text{threshold})^c + d},
\]

where \(c, d\) are real numbers computed during energy calibration and \(\text{energy}\) is the value after conversion to \(\text{keV}\) \[38\].

### 1.6 Source of data

As a source of data we used measurement from ATLAS Timepix3 detector network in CERN \[40\] (i.e. ATLAS-TPX3).

ATLAS \[36\] is one of the major experiments in Large Hadron Collider (LHC) at CERN. Wide range of physical experiments are conducted here, including the predictions testing of the Standard Model \[25\].

The ATLAS-TPX3 network is a follow-up project to previous Medipix and Timepix networks. They provided valuable information about luminosity, radiation field composition, radiation doses and material activation in ATLAS. Two pairs of Timepix3 devices were installed as shown in the Figure 1.8. We used sample data from the measurement of these devices from August and October 2018 within this thesis.
Figure 1.8: Positions of the Timepix3 devices in the ATLAS experiment [40].
2. Data processing and visualization

The path from the measurement of ionizing particles to creating information about pixel index, energy and time of arrival is described in the previous chapter. The general workflow overview is depicted in Figure 3.1. Here, we show possible ways how to process the low level data and appropriate ways how to parse them. Then the algorithmically most challenging part of connecting pixel events into clusters (bunches of pixels with spatial and time window requirements) follows. At the end of this chapter we will determine the most useful outputs of our clustered data that are suitable for monitoring during experiments with the detector.

Figure 2.1: Complete data flow from the Timepix3 chip to data analysis. White steps represent lowest hardware processing which is described in Chapter 1. Green steps are discussed in this chapter and the yellow ones are described in the following Chapter 3 and 4. Red boxes distinguish different software packages.

The specific nature of current measurement methods with Timepix3 presents challenging problems that need to be tackled. One of these particular properties of Timepix3 detector is the refresh rate of chip that allows up to 80 \( \text{Mhits/s} \) \(^1\) of data flow. Even with the currently used Katherine readout, which uses Gigabit Ethernet interface, the bandwidth limitation is around 16 \( \text{Mhits/s} \) \(^2\). Thus, fast processing is needed to get as close as possible to the theoretical maximum and, consequently, enabling near real-time data processing. With multiple Timepix3 devices (for example in a telescope layout \(^3\)) we could utilize relations between sensors to produce more detailed information. That implies the need for scalability in terms of number of Timepix3 devices connected directly to the clustering software.

In the principle of prototyped devices with fast paced development, modularity of this software should improve long-term sustainability. From the point of interest of physicists we are aiming to provide as much information about the running measurement as possible while doing non-trivial transformation of data for later use. All of the above mentioned fields of software requirements are discussed in this chapter.

\(^1\)Mega hits per second is a number of Timepix3 pixels read in one second. As it contains different informations than Megapixels used in camera chips, different notation is used.
2.1 Architecture

First of all we had to consider the possible technical approach to this framework. Constraints given by multiple factors, including the most prevailing mentioned in the beginning of this chapter, had narrowed the available programming languages and frameworks. Choosing the most suitable technical parameters of software was a crucial task for the long term development.

From currently available programming languages, C++ 14 offers balanced results in terms of speed and available libraries. With the modern elements of C++ we were able to avoid most of error prone mechanics, such as raw pointers (replaced with `std::shared_ptr` and `std::unique_ptr`) and multithreading (simplified with `std::thread`). As C++ 14 is currently well supported by all major compilers [5] (GCC 5+, MSVC 19.0+, Clang 3.4+) multi-platform compatibility was assured. Fast and reliable plotting tools were needed for graphic user interface as they had to retain the ease of portability between platforms. There are currently only few actively developed [14], cross-platform and all-round capable frameworks for C++, Qt was selected as the main GUI framework.

In this regard two approaches of plotting were tested before selecting a Qt chart module. The first combination was Qt [21] as a GUI framework with QtCharts [20] as the plotting library. The second was Qt as GUI with VTK [30] as plotting library. After testing, the second approach was discarded because of increased project complexity caused by the need to compile the whole VTK library with support of Qt [29], and inferior appearance. To achieve extensibility of our architecture, the software was divided into modules. Many of these modules run in separated threads. This multithreaded solution has bolstered the speed and the scalability in the number of devices and outputs.

The following paragraphs describe and explain the selected program architecture shown in Figure [2.2]. In this figure, the vertical direction symbolize the path from raw data (top) to the outputs and visualizations (bottom). Horizontally adjacent boxes represent parallel modules at a given stage of data processing. The red dotted blocks stand for each separate threads. The number of included modules and their types can vary across different real world use scenarios. With the chosen architecture, we can clearly differentiate between the backend logic and the interface exposed to a user. That led to the creation of two instances of the software. Both instances are composed of vital modules called internal modules and extension modules.

Console

For long term measurements or remote setting, omitting the GUI can be beneficial. We aimed to strictly keep the backend part of the framework separated from the user interface. Thus the first program was a pure console application. It was designed to support the usage on remote servers without GUI support or for batch clustering of already completed measurements. The console application can be run with command line arguments (with at most one Timepix3 device) or with settings from a config file (see Section [2.1.4]). All extension modules (more in Section [2.1.2]) in this version should operate without any need of GUI. This

\[2\] Graphical user interface
Figure 2.2: Pipeline of data processing. Data flow from the top to the bottom. Replaceable modules are at the levels of Feeders, Readers and Outputters. MTQueue denotes a multithreading safe queue for data transmission between the threads.
build target is independent on the Qt library.

**Graphical user interface**

For daily use and online measurement visualization, some form of GUI should be implemented. Therefore, the second build target was an application implemented with the Qt framework. We aimed to offer easy to use clustering and visual representation of the measurement data. Using this modular architecture, all internal modules (Section 2.1.1) of this GUI application are common with the console application. Moreover, some of the extensions (Section 2.1.2) included in the console version were used in the GUI application as well (e.g., *FileFeeder* reads the file as is and does not need to expose any information to the user).

### 2.1.1 Internal modules

Internal modules of the data processing software are designed to control data flow, create clusters from pixels and delegate results to designated outputs. We created a simple class *MTQueue* which is a thread safe queue for data to be processed in the following stages.

**Orchestrator**

As the whole architecture for data processing was based on modules, we concluded that some level of control over the process as whole can be useful.

The *Orchestrator* class is the base frame for the whole pipeline. It enables to select desired extensions (Section 2.1.2) that should be in a particular measurement. Afterwards it can start the processing (and stop it later).

**Clusterer**

Preprocessed pixels with mandatory parameters of *PixelData* (ToA, x, y) are grouped into clusters (Definition 2). In most cases we add ToT (Time over Threshold) to the *PixelData*, as it provides another substantial information. This component performs the clustering based on the index of device (this is important if multiple Timepix3 devices are connected simultaneously).

**Receiver**

Clusters from the *Clusterer* are pushed to the *Receiver* in the form of an array of *PixelData* and the additional variable *fineToA* (that represents arrival time of the first pixel in cluster). The receiver then dispatches a copy of these clusters to each of the registered *Outputter* extensions.

### 2.1.2 Extensions

With different input sources (e.g., Ethernet, USB, file) and different readout devices for Timepix3 (e.g., Katherine readout [39] and AdvaDAQ [49]) we aimed to simplify the development of various combinations of input sources and formats.

---

3ToA - Time of Arrival, x-coordinate and y-coordinate in Timepix3 chip
The same approach was applied on the output results. Long measurements and measurements with high data rate can benefit from filtering and storing specific clusters into files for further analysis. In contrast to this, short time measurements and testing require immediate monitoring of the situation at the detector. This led to the usage of different modules for each specific case. The procedure of enabling the usage of an extension is the following. Each extension must implement a parameterless constructor and the following two static functions:

- static std::string getName() – it returns the name of the extension,
- static std::string getDescription() – it returns a text describing the extension.

Based on the type of the extension (Feeder, Reader, Outputter), it must inherit the respective base class. Making the extension available for a use is then done by including an appropriate header file and extension class name as template parameter in AviableExtensionsConsole.hpp or AviableExtensionsGui.hpp depending on the type of the extension. When these requirements are fulfilled and the program is compiled, the extensions selected in the config file, are loaded as new modules to the data processing pipeline by the Orchestrator. Name of the extension in the configuration file must exactly match the string from the static std::string getName() method. It is recommended that this string is equal to the actual class name. Examples and more details can be found in Section 2.1.4.

Feeders & Readers

Before assigning individual pixels to clusters, we need to obtain the data from a source. We use the Feeder base class to manage the I/O communication. Its output in the form of lines representing either measurement info or (mostly) actual pixel data is interpreted in the Reader class. The Reader class is dependent on the specific format of the message and translates it into a format suitable for the Clusterer class. If present, energy calibration and timewalk correction can be used (see Section 1.5).

Outputters

Completed clusters are distributed to all registered instances of the Outputter classes. Each instance receives a separate copy of each cluster. The clusters are distributed separately to achieve simultaneous visualization and data storing in real-time (i.e. each Outputter has its own thread and copy of the cluster).

2.1.3 Lifecycle of components

Multiple steps are needed to create and control the user selected chain of modules in the Orchestrator. With the provided ConfigControl class and the TPXSettings struct, the user does not need to setup the Orchestrator manually and in correct order. We simply either fill up the TPXSettings with desired settings or simply point to the directory with settings and the ConfigControl.
will load the setting from the config file (details in Section 2.1.4). Next, the procedures inside the ConfigControl and Orchestrator are executed. They run after we prepare the TPXSettings. The first step in getting the measurement ready is to fill up the Orchestrator with instances of selected modules. However, some dependencies must be satisfied (e.g., the Reader must be added after the proper Feeder). We can either pass a shared pointer to an instance of the class or ask the Orchestrator to create the instance by calling a function with the class name as a template argument and, if needed, we can pass constructor arguments to the module. As there is a support for multiple devices, we need to specify for each module its ID, which corresponds to the ID of the chip. In Figure 2.3, we can observe the whole process of running and ending measurements. Components are started in reversed order as we need to follow the reverse path of the directed tree graph shown in Figure 2.2.

There are multiple occasions at which we want to end the measurement. The most obvious reason to stop the measurement occurs when all feeders reach their end state (e.g., UDP connection closes, end of file is reached). In this case, each Feeder signalizes its end state. In the case of continuous measurement we can end the process by calling End method on each input, this should be followed by properly closing the inputs (e.g., signalizing to Katherine readout to end measurement) and finalization of all buffered data (multiple buffers at different stages are finished).

As the final processing can be long term action (up to few minutes when there is a lot of unfinished data), we allow to abort the measurement. This process drops all buffers and calls the Abort method on each module. This procedure allows to handle a quick abort in a different manner than the usual end.

2.1.4 Configuration files

As each measurement can vary in terms of the number of the devices, calibration files and sources of data, config files can be used. We have chosen the INI file format as it offers simplicity and it is sufficient for the extent of settings in our software. The INI file format consist of sections (denoted as [section_name]) followed by one or more key-value pairs (in notation key=value). An example of such configuration file and mandatory properties of the configuration file are described next.

[0]
inputType = KatherineFeeder
inputFormat = KatherineReader
inputSource = 192.168.1.142:1555
outputWindows = WatchQt Integrate
ecalibDir = /Calibrations/K8–W0036/

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[index]</td>
<td>User defined index of the device, denoted as name of the section in the INI file, represented by whole number.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Example value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputType</td>
<td>FileFeeder</td>
<td>Represents the type of an input source of data readout chip. In this example we will read a text file.</td>
</tr>
<tr>
<td>inputFormat</td>
<td>KatherineReader</td>
<td>Sets the correct input source format. Katherine readout format is parsed to PixelData structure (more in Section 2.3).</td>
</tr>
<tr>
<td>outputWindows</td>
<td>WatchQt IntegrateQt</td>
<td>From all included Outputter classes, user can select multiple output windows. outputWindows can be selected as space separated values. WatchQt and IntegrateQt are both GUI extension for measurement visualization and statistics.</td>
</tr>
</tbody>
</table>

2.2 Input data sources

Providing raw data for parsing and clustering is the first part of the clustering pipeline. Custom classes for raw data must inherit the base class Feeder. Data from a source are referenced as separated records of a single pixel hit information to the Reader class. Currently, we have prepared four types of input data.

- FileFeeder is a source designated for offline clustering of data. It reads a whole file as fast as possible.
- FileFeederContinuous – In some measurements, it is preferred to use a different framework for data acquisition. Assuming that raw data are stored
in a text file and the file is continuously updated, we want to keep reading
the file until user interaction.

- **FileFeederActualSpeed** – To replay (simulate) acquired data (useful in
  short measurements), we read the file in accordance to individual time-
  stamps inside the file to simulate input at real speed of the measurement.

- **KatherineUDP** – Graphical user interface extension with basic control of the
  Katherine readout via UDP protocol. Data are captured directly from a
device.

### 2.2.1 Katherine UDP communication

The current Timepix3 readouts installed in ATLAS and MoEDAL experiments
are Katherine readouts (details in Section 1.4). We aimed to provide direct
acquisition from this device using UDP protocol over Ethernet. As of middle
of 2018 the standardized networking library in C++ have not reached general
availability state\(^5\). To sustain the general portability of the framework, we work
with the QtNetwork module of the Qt library.

The Katherine readout uses a two port design. One is for sending commands to
the readout while the second is dedicated to the data acquisition. The complexity
of the device\(^3\) and the number of available commands led to a solution, where
Katherine is set up with a specialized software. The measurement is afterwards
started (or stopped) with the presented framework. In future development we
plan in cooperation with the authors of the Katherine readout, to fully support
all commands and settings.

The commands are sent via 8-byte UDP datagrams. After each command, the
Katherine readout will send back an appropriate response (e.g., ID of Timepix3
chip). To illustrate the type of the commands, here we present those that are
used in our framework. The values used in the list are headers of the 8-byte
datagram and they are hexadecimal values representing the command ID.

- 0x01 – Set the acquisition time.
- 0x03 – Start the acquisition.
- 0x06 – Stop the acquisition.
- 0x0B – Get chip ID. This command is useful when probing the local network
  for available Katherine devices.

#### Pixel datagrams

Datagrams for the data-driven mode\(^6\) are in the form of 6-byte vector. To un-
derstand how the information about a pixel is stored, we need to cover two types
of vectors. The size of the UDP datagram is controlled by the readout and grows
with increased data flux. Based on the technical implementation of UDP protocol,
the maximum size of the UDP datagram is 65507 bytes\(^{28}\).

---

\(^5\)Status of ISO/IEC TS 19216:2018 \(^{13}\).

\(^6\)Modes of Timepix3 are described in Chapter \(1\).
The header part of the 6-byte datagram tells us the type of the packet encoded as a hexadecimal value. We are showing just two types of measurement data. As the Katherine readout supports all modes of the Timepix3 device (see Chapter [1]) the headers can be, for example, 0x7 to mark start of new frame or 0xC to denote end of the frame. But we are focusing on data-driven mode and thus we will show the details of the offset datagram (header 0x5) and the pixel measurement data (header 0x4).

Each UDP datagram starts with a 6-byte vector, which holds the time offset from the measurement start. The format is shown in Figure [2.4]. The vector consists of a header, zero 12-bits and a time offset. The time offset is an integer value that is used to compute the time of arrival for each pixel.

![Figure 2.4: The offset datagram is used to transmit the most significant part of the final time of arrival. The offset is an integer value.](image)

The pixel data are stored in a scheme described in Figure [2.5]. Should the variable localToA overflow, new time offset is sent. The header 0x4 is reserved only for the data-driven mode of the Timepix3. Followed by two 8-bit unsigned integers, we obtain the x and y values. This limits the maximum number of the pixels to 65536 (i.e. with square chip to 256 × 256). The remaining 28-bites of data are reserved to the time of arrival and the time over threshold. The time over threshold is 10-bit unsigned integer value. The time of arrival is divided into a localToA which is added to the offset and a fineToA that is the least significant part of the time variable. This split is caused by the internal clock speed of the device where the localToA clock is running at 40MHz and the fineToA at 640MHz.

![Figure 2.5: Data-driven datagram for one pixel hit. Contains x and y pixel matrix index, timestamps and time over threshold of the pixel.](image)

The final time of arrival is then computed as:

$$ToA = (offset \ll 14) + localToA$$

Where $\ll$ denotes bit shift operator. As the high precision mode of Timepix3 has additional precision variable fineToA we will store it separately and use its value in the following section. The matrix index is obtained as:

$$Index = Y \times 256 + X$$

Following this datagram structure, we obtain all the necessities – Index, ToT, ToA and fineToA. This rather complicated notation of arrival time is caused by
the nanosecond precision of Timepix3 chip and the goal of Katherine readout to permit long time measurement. The next section will explain these values.

2.3 Katherine file format

Data used and available from ATLAS-TPX3 experiment are currently measured and stored mainly in the Katherine output format. Thus, we have focused on this particular format, while preserving the possibility to create a new Reader derived class for different formats. For the purpose of data processing, we will describe just the data-driven mode outputs. Each record consist of the following:

- **Index**: Index in the range \((0, 65536)\) to matrix \(M_{256,256}\). Represents \(X\) and \(Y\) pixel index.
- **ToT**: Time over Threshold which can be translated to estimated energy (in keV) when calibration files are used (details in Section 1.5).
- **ToA**: Time of Arrival as a multiple of 25 nanoseconds.
- **fineToA**: Fast Time of Arrival as multiple of 1.5625 nanoseconds.

To compute the precise time of arrival in nanoseconds (from the beginning of the measurement), we use following formula:

\[
\text{TimeFromBeginning} = 25 \times \text{ToA} - 1.5625 \times \text{fineToA}
\]

For sake of simplicity, in the rest of this theses, we will use \(\text{ToA}\) as the time from the beginning of a measurement. Above explained technical difficulties are omitted as they are negligible for understanding the clustering.

2.4 Clustering

Until this point we were still working with a stream of discrete pixels. Our target in this section is to make use of the time precision and spatial information to connect the individual pixels into events. The data-driven mode brings more information (opposed to the frame based ToT mode in the Timepix) at a cost of more complicated data processing. In a frame based mode, we receive a matrix with values integrated over a selected time window as shown in Figure 2.6(a) (a similar principle as in a photographic camera where we have exposure time and accumulated charge in each pixel).

To simulate such behavior we could create a matrix containing energy for each pixel within a given time window. We can compute such matrix based on the time of arrival and the time over threshold, but that would bring back two issues present in older Timepix devices. Firstly, it would cause event overlapping and secondly, we would have to find a way how to choose a correct time window. At this point, we start to work with the term cluster. A cluster is a set of pixels representing one event.

\footnote{For example 32-bit integer is capable to hold only values in magnitude of seconds}
Knowing the speed of particles and the charge drift speed (discussed in Chapter 1), we can determine the maximum time an event can take (i.e. time difference between time of arrival of the first pixel and the last pixel in cluster). Based on the measured time and spatial information we present a fast algorithm for cluster creation. Before presenting the actual algorithm, we will define some key terms and then we will show that we are able to create separated clusters of pixels that represent individual incidents as depicted in Figure 2.6(b).

Figure 2.6: Cluster versus integration. Source of data is ATLAS experiment. Black background in (a) represents zero energy pixels and the whole matrix $M_{256,256}$ is displayed. For clear view, in (b) zero energy pixels are white and all non-zero energy pixels are represented using the color scale as in (c) and the image is cropped to matrix $M_{80,80}$.

**Definition 1** (Pixel neighbor). Let $p_1, p_2$ be two pixels in a 2D grid.

- Pixels $p_1, p_2$ are 4-neighbors if they are adjacent horizontally or vertically.
- Pixels $p_1, p_2$ are 8-neighbors if they are adjacent horizontally, vertically or diagonally.

Figure 2.7: Type of pixel neighbors. Yellow pixels denote neighbors of the red center one.

**Definition 2** (Cluster of pixels). Let $\Delta t$ be a positive real number called size of the time window, $X$ be a set of pixels and let for each pixel $p \in X$, $T_{OA_p}$ denotes its time of arrival. A subset
$C$ of $X$ is called cluster, if for each pair of pixels $p, p' \in C$ the following two conditions are satisfied:

1. $|\text{ToA}_p - \text{ToA}_{p'}| < \Delta t$, i.e. the times of arrival of the pixels $p$ and $p'$ differ less than $\Delta t$.

2. There exists a path of pixels in $X$ from $p$ to $p'$, i.e. a sequence of 8-connected pixels in $X$ which starts in $p$ and ends in $p'$.

There are different approaches how to create clusters from a stream of pixels. As the stream of individual pixels is not necessarily sorted by the time of arrival we have to adjust the algorithm for that. There can be delay (due to the readout electronics) up to $D = 200000$ ns. The simple method would be to sort the data after the measurement and connect pixels into clusters regardless of their spatial relation. This would be sufficient for low intensity sources. To create clusters in real-time and with spatial information included we are going to construct the clusters from individual pixels in real-time.

The idea for processing new pixel can be summarized in these points:

- Keep a set of partial clusters (denoted as $openCLusters$)
- Try adding the new pixel to an existing cluster.
- If there is no existing cluster to which the new pixel can be added, create a new cluster consisting just of the current pixel.
- If the new pixel can be added to more than one cluster, join them.

Using the following algorithm, we create new clusters or add pixel, to an existing cluster and eventually join clusters, if needed. Moreover, if some of the partial clusters contains only pixels with ToA less than ToA minus $D$ of the new pixel, then such cluster could not be extended anymore and can be outputted and removed from the set of partial clusters. In this section we introduce $timepixel$ as a term describing pixel coordinates ($x$ and $y$) and ToA.

**Algorithm 1** Process one pixel

**Require:** $openCLusters \leftarrow []$

1: **procedure** PROCESSPIXEL($timepixel$) \triangleright Pixel containing coordinates and ToA

2:     $added \leftarrow \text{False}$

3:     **for each** cluster in $openCLusters$ **do**

4:         **if** $CanBeAdded(cluster, timepixel)$ **then**

5:             **if** $added$ **then**

6:                 $lastCluster \leftarrow \text{JoinClusters}(cluster, lastCluster)$

7:             **else**

8:                 $AddPixel(cluster, timepixel)$

9:                 $added \leftarrow \text{True}$

10:                $lastCluster \leftarrow \text{cluster}$

11:         **if** $\text{not} \ added$ **then**

12:             $lastCluster \leftarrow \text{CreateNewCluster}(openCLusters)$

13:             $AddPixel(lastCluster, timepixel)$

14: **CloseAndDispatchOldClusters(openCLusters)**
Let each cluster have a quadtree data-structure *neighbors* representing adjacent pixels to current pixels in the cluster. The cluster is then represented by a list of pixels in the cluster and quadtree with neighbors. We are able to assign pixels to clusters using this algorithm. But how do we determine, if a pixel can be added to a cluster? First of all a pixel must be 8-connected to the cluster and must belong to the time window of the cluster. We chose to store all neighboring pixels for each cluster.

Choosing the right data structure to store the neighboring pixel: if we would store the neighbors in an unsorted array then the complexity to check if pixel belongs to the cluster would be linear (i.e. we would need to check all of them). On the other hand if we would store neighbors in same size matrix then each cluster (even with only one pixel) would take up at least 65536 bits. To determine if a pixel is neighbor without scanning the whole matrix \( M_{256,256} \) (i.e. to get response in logarithmic time) we use a tree data structure known as quadtree (Subsection 2.4.1). This structure offers balanced performance between memory used and time to check if pixel is neighbor. Detailed description is in the following subsection. When we add pixel to the cluster we must append new neighboring pixels.

**Algorithm 2** Pixel adjacency check

1: function CANBEADDED(cluster, timepixel)  
2: \[ \text{pixel} \leftarrow x \text{ and } y \text{ coordinates of timepixel} \]  
3: \[ \text{node} \leftarrow \text{cluster.neighbors} \quad \triangleright \text{We start with quadtree root} \]  
4: if pixel not in node.boundaries then \[ \triangleright \text{Pixel is out of root boundaries} \]  
5: return false  
6: while node not leaf node do  
7: \[ \text{find quadrant in node where pixel belongs to quadrant} \]  
8: \[ \text{node} \leftarrow \text{quadrant} \]  
9: if pixel in node then  
10: return true  
11: else  
12: return false

The function \( \text{CreateNewCluster(openClusters)} \) creates a new cluster with empty set of pixels and with empty quadtree neighbors.

The function \( \text{CloseAndDispatchOldClusters(openClusters)} \) finds clusters beyond the current time window (i.e. no new pixels can be added because the time difference is to big and there cannot exist another pixel belonging to the cluster), deletes now unnecessary quadtree *neighbors* and sends the cluster to the output.

**Algorithm 3** Join clusters

1: function JOINCLUSTERS(clusterA, clusterB)  
2: \[ \text{leaves} \leftarrow \text{all leaves from clusterB} \]  
3: \[ \text{pixels} \leftarrow \text{all pixels from leaves} \]  
4: for each pix in pixels do  
5: \[ \text{AddPixel(clusterA, pix)} \]
Algorithm 4 Add pixel to cluster

1: procedure AddPixel(cluster, pixel)
2:     Append(cluster, pixel)
3:     eightNeighbor ← pixels in 8-neighborhood of pixel
4:     for each newNeighbor in eightNeighbor do
5:         Add(quadtree.neighbors, newNeighbor)

We are adding as neighbors even pixels that are inside the cluster. This is not an issue because the maximum duration of the cluster is lower than the dead time of the pixel. Before passing the cluster to the Outputters we also remove the neighbor’s quadtree from the cluster.

The only remaining task is to determine when clusters from the openClusters shall be closed and dispatched to the Receiver. This is determined by the properties of Timepix3, where we can get pixels with up to 200 microseconds long delay.

2.4.1 Quadtree

Quadtree is one of the currently used tree data structures in 2D graphics and other two dimensional problems published in 1974 in the following article [44].

It allows to store efficiently a set of points in two-dimensional space. The tree nodes have up to four children. Points are stored in leafs. For sake of simplicity, each leaf covers area of $1 \times 1$. Let us have root node representing two dimensional square area and a given set of points in the same area, then the recursive definition of quadtree is the following:

1. divide the current area (i.e. the parent node) into four quadrants,
2. if a quadrant has a point in it, add a child representing the quadrant to the parent node,
3. if the area of the quadrant is at least $2 \times 2$, run recursion for each child, otherwise the node representing the quadrant will be a leaf node representing one point.

In pictures, quadtree can represent square quadrants of an image and recursively its sub-quadrants (e.g., see Figure2.8(a)). In our program we use quadtree to store the possible neighbors of a cluster. In contrast to the above definition, leaves of our quadtree can store points in an area of dimension $2^k \times 2^k$. 

26
To determine the ideal leaf size in our implementation, we ran two tests:

- speed – we measured the time to cluster a real dataset (i.e. the value of $k$),
- memory usage – we approximately measured average memory usage of the cluster.

To measure time, we have tested five times each leaf size (all powers of two) and averaged the results. In Figure 2.9 we can see speed increase from leaf size 2 to 16 then the time values are stabilized and are rising when we use whole matrix. The memory usage test was an approximation ran on the same dataset. We measured only the sum of bytes used in each leaf of the cluster and then averaged across the dataset (we can assume that the overhead of the tree structure is smaller by one order compared to the data in leaves). Results are shown again in Figure 2.9.

We can observe an exponential increase of memory usage with increasing leaf size. Although the fastest speed was at leaf size 128, the memory usage between sizes 16 and 128 increased over 100 times. Thus, we have chosen the leaf size $16 \times 16$ as the most suitable, despite the fact that the leaf size $128 \times 128$ was slightly faster.

The data used for testing was taken from a real measurement in ATLAS experiment to simulate real-world usage. For special experiments where we anticipate that clusters with size smaller than 8 prevail, we can lower the leaf size. This is possible because clusters begin with quadtree covering the area with size of leaf and as we join clusters or add pixels we rebuild the tree (we start with a leaf and build up the tree if we exceed the leaf boundaries). The above definition of quadtree is illustrative to show how the tree works when we already have large cluster.
For the best performance, leaves of this tree are boolean bit-arrays, representing $16 \times 16$ square sections of the chip matrix. Thus, the maximum number of steps through the quadtree to determine if a pixel is a neighbor is $\log_2 \left( \frac{256}{16} \right) = 4$ plus one comparison operation to check the position in the leaf bit-array. This can be often lower, as we prune the tree. Branches with no marked pixels are cut at top-most possible level.

Let us assume the quadtrees are aligned (2D space is divided into cells of the same dimension) and we want to join two quadtrees. Joining two trees is done by following steps:

- we start by determining which of the tree is smaller, let the smaller one be called $S$ and let the other be called $B$,
- we add each leaf $L$ of $S$ into $B$,
- if leaf $L$ is in the pruned branch, we will recreate that branch and add the leaf into $B$,
- if the leaf is out of bounds of the quadtree, we recursively add new root to the tree in the correct direction from the old root until we are able to add the leaf.

2.5 Monitoring and storing results

We have transformed the incoming pixel data into clusters. This allows us to create an innovative visualization of the running measurement. As shown in the previous subsection, with pixels alone we can perform the integration over time. But with clustered data we obtained significantly more information.

We now have separated and time sorted clusters, which was not true for pixel only data. We combined the pixels into clusters and thus we are able to display the clusters one by one. The cluster has information about its total energy as well.
as its size (number of pixels). With the given structured data, we can now provide more in-depth statistics and we will show the possibility of real-time classification and filtering of clusters. During tests and measurements with Timepix3 it can be useful to visualize physically interesting statistics.

For these purposes three different Outputter extensions were created. These modules aim to provide a real-time overview and functions in the following areas:

- Time statistics – In many experimental tests, ionized particles are directed to the detector in bunches. This can be observed as peaks in energy fluxes of the detector.

- Source histograms – Different sources of the ionized particles have various energies and affect different number of pixels. Therefore, a histogram of the cluster energy and a histogram of the cluster size are beneficial.

- Integration window – Simulate the integration process and observe what is happening at the sensor as a whole (e.g., to check the alignment of the chip in a particle beam).

- Real-time storing – While observing the above mentioned we should store the clustered data directly to a file. Storing clustered data simplifies the subsequent analysis of the data.

Given that we already have the pixels in the clusters, we should capitalize on the additional information. This led to implementation of online filters. Furthermore, we have created the architecture in such a way that the filters can be dynamically loaded, even during an ongoing measurement.

### 2.5.1 Filtering plugins

Before describing the actual outputs of our framework we will present one common part of all currently included outputs. Basic information that each cluster provides without any further processing is:

- the size of the cluster in pixels,
- the total energy of the cluster in keV,
- the energy of each pixel in the cluster and the time of arrival of each pixel and
- the time of arrival of the earliest pixel in the cluster.

We can use these properties to filter the clusters that are processed by the Outputter. We are keeping all Outputter classes separated, so that each output receives all clusters and the filtering is performed in individual Outputters separately.

The most straightforward type of filter is based on the size and the energy of the cluster. As it is mainly used for a visualization, the filter is implemented as an Qt Widget. In Figure 2.10 it is shown how the widget looks like and implicitly

---

8Assuming we are using energy calibration, for details see Section 1.5.
we can see its usage. By setting the lower and the upper bound of the energy or the size we can adjust the clusters that are processed (e.g., if the cluster is included in a flux statistics).

Figure 2.10: Filter widget for size and energy clustering. The widget is a part of the output, see the following sections. The range of the values is from zero to the maximum value during measurement. By moving the right slider to the end we can set the maximum value to infinity.

There is one more important aspect of this architecture. In the following chapter, we will discuss possible ways how to classify clusters. To translate the working classification methods tested in Python into this framework we have prepared the following plugin architecture.

As the real-time processing is more performance dependent in contrast to the offline approach, the methods have to be rewritten into C++ from Python. The only requirement for the plugin is to use Qt plugin architecture. Why have we chosen the Qt plugin architecture?

The goal was to create plugins as dynamic libraries. That is beneficial in two ways. Firstly, we do not have to build the whole project each time again. The second is the possibility to load the filter only when it is needed. But there was a direct conflict between keeping most of the code library independent (i.e. not using the Qt library where it is not needed and stick to pure C++14) and keeping the framework multi-platform. In conclusion, we have decided that the benefits of Qt prevails. It offers a platform independent plugin loading interface and using the CMake build system, appropriate dynamic library format is always created. The simplicity of creating a plugin is preserved and the Qt library is already heavily used in our framework.

Apart from the size and the energy filters we have prepared two example plugins. The first one is showing the creation of a plugin with the Qt widget, including a simple form of a control interface. The second one does not have any visualization. Both cases loaded into the framework are shown in Figure 2.11. Both example plugins are available in the source code of the framework.

Important steps to create custom filter are the following:

- Inherit the Plugin class and in the case of a GUI filter, inherit the QWidget as well.
- Override the method bool Satisfies(ClusterData).
- Follow the general pattern of creating a plugin for Qt [20]. The example plugin code can be used and reused with minor changes.
The filter plugin with Qt widget uses the standard Qt UI files and can offer control over the filter. In the given example, we have created a simple filter for predefined base categories.

Figure 2.11: Example of loaded filter plugins.

With the presented design we can quickly create new filters for all platforms. These plugins can be created independently on the main framework. In the future, the goal is to filter desired events directly during the measurement based on the subject of the test.

### 2.5.2 Event statistics overview

One of developed outputs is a statistics overview window. With replayed data we can see in Figure 2.12 possibilities of this `Outputter`. This extension allows us to see three different graphs representing the current measurement.
Figure 2.12: Event statistics output. In the left column there are controls of individual graphs. In the middle part, there is a space for up to three filters. In the right section we can see the produced graphs. Each filter represents one colored line in the charts. The size and energy filters have the color displayed in the control panel. The yellow color is representing the basic class filter. The settings can be hidden with the top left “hamburger button”. Replayed data are from a measurement at Super Proton Synchrotron experiment in CERN.

To get an immediate overview of the current flux, we have created the top graph. Its purpose is to show the number of clusters (i.e. events) in a given time window. Inside the time window we can set the time resolution. Lowering the resolution (i.e. the value time resolution in Figure 2.12 is higher) leads to fewer points in the chart. The number of cluster is not normalized (with lower time resolution we do have more clusters). The y-axis is simply the number of clusters per resolution time.

The middle graph represents histogram of the cluster energy. On the x-axis there are the bins of energy (in keV) and on the y-axis we have the number of clusters per bin. In this graph, we can adjust the value range and the granularity of the histogram. The granularity affects the size of the bins. All bins have the same size.

The bottom graph is again a histogram. The graph shows the distribution of the cluster size. We can modify the range of the histogram. In the previous subsection, we explained the filters for the outputs. In this overview, we imple-
mented the filters as individual plot lines. By adding a new filter, we create a new line in the graphs. This allows to observe simultaneously different parts of the event spectrum.

### 2.5.3 Cluster integration window

The second visualization extension that was developed is the integration window. This window aims to provide a visual representation of the sensor matrix in the selected time window. We present two 256×256 pixel matrices. The upper canvas draws clusters with color representing the relative time of the individual pixels. We have to compute the time delta between clusters in the given time window.

\[
\Delta \text{ToA} = \text{ToA}_{\text{newestCluster}} - \text{ToA}_{\text{oldestCluster}}
\]

Given this information we are able to use a colormap to find the correct color to represent the relative time of arrival.

A more straightforward approach is used to showcase the energy of each pixel. The second canvas accumulates the energy of individual pixels in the given time window. If two clusters are overlapping the energies of covered pixels are summed.

The actual window of the measurement with a time window of 100 seconds with applied filters is shown in Figure 2.13. In this output, we have used the filters in a bit different manner than in the TPX event statistics window. In this case, the filters are stacking, that means the cluster has to satisfy all filters to be shown. We can set the filters at any point during the measurement and immediately see the difference. An example of this behavior is shown in Figure 2.14. Both canvases were captured at the same moment and just by setting the minimum cluster size to 40 we can clearly observe the biggest events of past 100 seconds (in terms of size, similarly we can adjust the minimum cluster energy). Thus, we have used the advantage of clustered data to improve the real-time visualization capabilities.
Figure 2.13: Integration window. With the first column we can adjust the time window and refresh time of the window. With the refresh time bigger than time window, we will possibly skip (not see) some of the clusters. Middle column is used for up to three filters which are substractive (cluster must satisfy all of the filters). Right part of the visualization shows ToA and ToT canvases. Color bars and their colormaps are automatically adjusted to cover whole range of values in canvases. The settings can be hidden.
Figure 2.14: Comparison of the 100 seconds long measurement with and without filtering involved. On the left we kept all of the clusters captured during the integration time. On the right we can see only clusters with the size bigger than 40 pixels.

Colormap

For displaying the ToA and energy level in the integration window we use color mapping. This process is used to create (from a set of colors) a full color gradient in a given range. If we then request a color from the colormap range we will get the correct color interpolation in the gradient. We have implemented the following color mapping class in our framework.

We initialize the colormap with at least two colors represented as `QColor` objects in the RGB\(^9\) colorspace. The colors are evenly spread over the colormap. However, the interpolation between colors is more convenient in the HSV\(^10\) colorspace. To switch between the colorspaces, we use the internal Qt library transformation (the process is shown in \[11\]). In the HSV colorspace, we can then directly interpolate numerical values with good results. The colors should not be achromatic (i.e. grayscale) because that yields zero hue value.

---

\(^9\)RGB – stands for red green blue color notation. In our case each color channel is 8-bit integer.

\(^10\)HSV – hue, saturation and value color space. It represents more closely how humans perceive colors.
Algorithm 5 Get RGB color from colormap

1: procedure Get RGB(value)
2: ▷ value should be in colormap range
3: ▷ We need to find the two closest colors to interpolate between them
4: ▷ We have min and max values representing colormap range and we have numColors of colors.
5: step = (max - min)/(numColors − 1)
6: firstColorIdx = ⌊(value − min)/step⌋
7: secondColorIdx = firstColorIdx + 1
8: coefficient = ((value − min) − (step * firstColorIdx))/step
9: hsvStart = GetHSV(firstColorIdx)
10: hsvEnd = GetHSV(secondColorIdx)
11: ▷ We will now perform the interpolation
12: hsvResult = hsvStart ∗ (1 − coefficient) + (hsvEnd ∗ coefficient)
13: return ConvertToRGB(hsvResult)

2.5.4 Saving the clusters

To avoid storing the raw pixel data we prepared the Outputter for saving data. The user interface is shown in Figure 2.15.

Figure 2.15: User interface for saving the clustered data. In the left part, we can choose directory to save the results (if it is empty, the current directory of the program is used). To avoid overriding files, the filename is automatically generated based on the current time. In the second part there are possible filters. The user can store filtered and unfiltered data simultaneously (two files are created).

We allow simultaneous saving of all processed clusters and filtered clusters.
Filters are used in the same way as in the previous subsection (i.e. they are stacked). We provide a simple statistics of the number of processed clusters and clusters that passed the given filters.

Currently, we support saving data in text format in comma and semicolon separated format. Each line of the file represents one cluster in the following format:

```
clusterID, firstToA, size, pixel, pixel....
```

Where `clusterID` starts from 0 for each measurement and is used to have correspondence between filtered and unfiltered files. The `firstToA` value is the time (in nanoseconds) of arrival of the first pixel in the cluster in a floating point notation. The `size` is the amount of pixels in the cluster. These values are followed by `size` (the amount) of pixels in the format:

```
[ x; y; time of arrival; time over threshold ]
```

Where `x` and `y` are pixel matrix indices in range from 0 to 255. Time of arrival is again kept as a floating point number of nanoseconds from the start of the measurement.

For more efficient storage space usage, we can use a compressed format. In the future, we can improve the write speed and optimize the space by using a binary format. As the Katherine readout supports measurements in magnitude of days while keeping nanosecond precision, we may in the future use a similar process of division of time into offset and local time of arrival to support these really long measurements. Currently we have tested measurement length up to hours.

### 2.5.5 ROOT files

In the early stages of designing this framework we started with an idea to build up this software on the ROOT framework \[22\]. The ROOT software was created in CERN. Its main goal is data processing, in particular for physicists. The framework contains C++ interpreter with strong base of data analyzing and visualization tools. ROOT introduces its own binary format for data storing.

The binary format allows compressed saving of C++ objects. It includes classes descriptions thus allowing C++ objects recreations. The files are organized in a tree structure that is efficient with huge data amounts that prevail in experimental measurements. ROOT also ensures compatibility between architectures (e.g., little or big endian).

Physically a ROOT file has a header followed by records with variable length. Each record starts with 4-bytes that represents record size in bytes, if the value is negative then the record is marked as deleted. Logically the file is structured in similar way as UNIX filesystem. It contains directories with possibly unlimited depth of subdirectories. Default behavior of ROOT is to compress every object inside the ROOT file.

Unfortunately, after implementing support for this binary structure in the console version we have came across three major obstacles that prevented building up on this format.
First inconvenience is necessity to include another rather big library to the project. The combination of multithreaded architecture and ROOT has proven to be cumbersome with older versions of ROOT [37] and that conflicts with our multithreaded architecture. Finally, by using our own CSV format we have made the use of outputs easier as can be seen in the next chapter. We have kept the working Outputter class that outputs clusters directly into ROOT files in our project, but without GUI support, this extension is omitted from the standard build setup. In the future with new ROOT version this format can be investigated once again. An alternative way of creating ROOT output files is to create single-purpose utility to convert our CSV files into the ROOT files.
3. Timepix3 cluster data

After going through the process of getting the data from the device, clustering them and saving them, we will now focus on the analysis of the actual clusters. First of all we need to read and visualize the clusters. After that we will prepare the environment for the testing of the features. Then we will discuss the limitation of the data and what we would like to obtain from the analysis. In the end of this chapter we will examine the properties of the clusters. We will involve a non-trivial description of clusters to have a greater chance of finding and labeling the exotic events.

![Data Flow Diagram](image)

Figure 3.1: Complete data flow from the Timepix3 chip to data analysis. White steps represent lowest hardware processing which is described in Chapter 1. Green steps are discussed in this and following chapters and the yellow ones are described in the previous Chapter 2. Red boxes distinguish different software packages.

Although we have already created the framework for real-time acquisition and clustering in C++, we had to consider completely different aspects of data analysis program. Opposed to the software presented in previous chapter, we are not focusing on real-time performance nor live statistics. Our first goal was to create clustered data browser with visualization. Next task was to provide a convenient tool for an annotation and feature visualization of clusters. For this purpose we will consider and test different artificial intelligence approaches.

3.1 Cluster viewer and analysis software

Assuming the above mentioned differences between the clustering and classification software, we have chosen to divert from the C++ programing language. We selected Python as the environment for cluster analysis. The main reasons behind this decision were two. Firstly, Python offers broad scale of support libraries, such as scikit-learn and NumPy.

With these libraries we were able to focus on the actual classification methods and their comparison without re-writing already known methods. The second reason was faster development with Python especially without the necessity to compile. Python is also a less error susceptible language. Python has dynamic memory allocation with garbage collection, in contrast to the raw pointers in C++. Another Python advantage is the fact that it is dynamically typed language (types of variables are determined during runtime).

To build up the graphical user interface, we used the capabilities of the Qt library (see Chapter 2) in the form of PySide2 wrapper library. This project
is based directly on Qt and provides bindings to the underlaying C++ library. It is currently supported directly by the Qt Company [24][4] which ensures compatibility and long term sustainability. Using the PySide2 we can benefit from the knowledge of the principles of the Qt library and we can quickly develop a GUI in Python. Moreover, the resulting GUI similarity to the framework described in Chapter 2 is an unintended benefit. For plotting and interacting with the displayed clusters we use the Matplotlib library [15]. PySide2 provides a convenient way of including Matplotlib canvas into the GUI of the application [9].

Figure 3.2: The architecture of the Data Browser. The clustered data are read from a CSV file and parsed using the CSV File Reader. The Data Browser controls the file reader (e.g., read next, seek to index, read previous) and passes clusters to further processing. The ToA and ToT graphs are displaying the current cluster with colors representing energy and difference from the first time of arrival in cluster respectively. The 3D ToT graphs allows optional 3D visualization where the third dimension is the energy in keV. The experiment visualization component is meant for feature testing and has its own canvas.

Figure 3.2 presents the architecture of our analysis software. The processing is simplified opposed to the real-time framework presented in Chapter 2. We open the clustered file for reading and then we visualize the cluster or test features. Individual blocks of the software are described in the following sections.

3.1.1 CSV file reader and data browser
To read a CSV file we created a simple class that allows basic manipulation with the cluster file. The parsing of the file is an integral part of the reading. The given file format was described in previous chapter.

For fast navigation in the clustered data file we implemented the following operations:

- set minimum size of cluster,
- read next – read the following cluster that satisfies the size criteria,
- read previous – by storing the position in file we are able to seek quickly to the previous shown cluster\(^1\) and
- seek to ID – seek to the cluster with given index or the nearest if not found.

\(^1\)We are keeping the seek positions in the memory for fast backward search. The complete sequence of seek positions of all visited clusters is stored.
3.1.2 ToA and ToT visualization

To display the actual cluster, we present two cluster representations, shown in Figure 3.3. This visualization brings the ability to see the relative time of arrival and thus recognize the relative drift time of the pixels. With the energy visualization we can observe the energetic core of the event and eventually filter out the unwanted (and yet not fully understood) halo effect around it.

![Time of Arrival and Time over Threshold](image)

Figure 3.3: Visualization of the cluster using Matplotlib. On the left side there is a cluster with color representing the relative time of arrival. The other image shows the energy deposition in keV at individual pixels. Image is zoomed into the part of the 256 × 256 pixels chip where the cluster appeared.

3.1.3 3D visualization

An alternative visualization is a projection of the energy into the third dimension. This can bolster the ability to investigate the event, for example to better see the volcano effect on Timepix3 chip, that occurs with high energies [46].

![3D Visualization](image)

Figure 3.4: 3D visualization of the energy of a cluster.
3.1.4 Feature testing and manual labeling

Before we begin with the classification of the clusters, we need to investigate the properties of individual clusters. As the result we want to determine well describing features of the clusters. This process can involve computation of derived characteristics from the basic cluster information. With bigger cluster some form of image analysis can find a use. To test these properties we implemented additional canvas into the analysis software to observe the tested feature immediately. From the ATLAS and MoEDAL experiments (see Chapter 1) we have unlabeled data. Physicist can label some of the data (as it is described in the following Section 3.2). For that, we provide a function to define and label the clusters viewed in the analysis software. When manually labeling, the data are stored in a separate file in the format:

\[ \text{clusterID, feature}_1, \ldots, \text{feature}_n \]

3.2 Classification goals

To choose the most suitable method for the cluster classification we need to begin with a description of the data. Generally, clusters coming from the measurements are unlabeled. Typically, some labeled data can be obtained from simulations. There exist methods for particle physics simulation, such as Geant4 [19]. However, the Timepix3 reponse has not yet been implemented to full extend. Another difficulty is to find out what the actual cluster describes, as it can be proton, delta-ray or even particle decay. We will show in the following paragraphs the possibility of manually categorizing the clusters into six basic categories. That will allow a partial usage of semi-labeled classification methods. Using the categories as metrics of the measurement (proportion of each category) will bring physically useful statistics. Ultimately, we would like to isolate exotic particles and events from the data that can be later examined individually. Methods with variable amount of classes or with the possibility of changing the granularity of classes should help with classification of the unlabeled data.

3.2.1 Basic categories of clusters

As we mentioned above, we can manually separate clusters into six categories, as shown in Figure 3.5. We will now describe informally the categories and propose features that can help to identify them. Detailed representation of the proposed features representation will follow in Section 3.3.

For the smallest clusters which have one to four pixels we use the term **dot**. As the particle does not always hit directly only one pixel but border between up to four pixels, we consider the dot to be between one and four pixels with maximum width and height of 2 pixels (i.e. for pixels in row are not considered as a dot). Most significant attribute of this class is its size. It usually represents photons and electrons.

**Long gamma** is representing again photons and electrons but with prolonged track with more than four pixels which is caused by the angle of the particle at which it traverse the chip. These tracks are limited by mean energy and size.
Figure 3.5: Six categories of known cluster types.
Heavy ionizing particles are the source of two types of clusters. The first is a heavy blob that is created during nearly perpendicular flight. These clusters have most energy deposited in the center of the track and are accompanied by large (in terms of pixels) halo. The cluster is roughly round.

If the angle, from which the particle is coming, is steep we can see heavy track. The track is straight and bulky with higher energy mean and higher total energy. Its roundness is much smaller and the largest halo effect is at one end of the cluster.

Minimum ionizing particles (such as cosmic muons 17 23) are creating straight tracks. Their main feature is the maximal pixel energy under 200keV and average energy below 50keV. The halo effect is not present and the width of cluster is usually lower than 4 pixels. Compared to the long gamma, straight tracks are longer with higher total energy.

Energetic electrons are a source of curly tracks, these tracks can be connected to a heavy ions clusters as a representation of delta electron 8. As they share similar significant properties as straight tracks, we should create the feature that describes the winding of the cluster.

3.2.2 Beyond defined groups

If we were able to categorize the particles in the six categories we would be left with unknown clusters that can be potential records of exotic events. Such events can be for example heavy ions with outwards spreading delta electrons or a particle decay. An example of a decay can be seen in Figure 3.6. The amount of such clusters is below one percent of all recorded events as we will show in the following chapter. As we are unable to correctly find and label these data we have to find an appropriate description of features and then select suitable unsupervised classification methods.

\footnote{Winding is a term used to describe how curly or non-collinear the cluster is.}
3.3 Feature engineering

For each cluster we possess a list of time pixels (with respective $x$ and $y$ coordinates). Each pixel has its time of arrival (in nanoseconds) and energy (in keV). We will now describe additional information that we are able to obtain from these basic information and that we can use as features for classification.

3.3.1 Feature selection

The simplest feature that we can use is the actual **cluster size**, that is the number of pixels in the cluster. The **total energy** of a cluster expressed by sum of energies of its individual pixels is another feature. A directly connected value is the mean energy:

$$\bar{E} = \frac{\text{total energy}}{\text{cluster size}}$$

To distinguish heavy tracks from heavy blobs we introduce the **roundness** of a cluster. It should represent the area of a circle in comparison to the particle. We find two most distant pixels in cluster (we use euclidian distance) and we compute the distance $D$ between them. Then the roundness is computed as:

$$\text{roundness} = \frac{\text{cluster size}}{\pi \times \frac{D^2}{4}}$$

and it describes the ratio between the event area and the area of a circumscribed circle.

The distinction between curly and straight tracks is hard to find using the above characteristics. We would like to describe how much is the cluster winding. There are possible multiple approaches to rank the winding. For example

![Figure 3.6: Exotic event representing decay of a particle](image-url)
computing the convex hull of the cluster. But we struggled with the precision of the winding measurement and with the normalization of the convex hull size as shown in Figure 3.7.

Figure 3.7: An example of computing the convex hull on three functions. From left to the right: $\sin(\pi \times 0.8x), \sin(\pi \times 2x)$ and $\sin(\pi \times x)$. The area comparison is: 89.4%, 100% and 89.5%. Although we can immediately see the difference between the winding of individual functions, we are unable to correctly distinguish them using the convex hull area.

We have instead decided to use fitting method. We use an enhanced polynomial fitting method with an example result shown in Figure 3.8. Polynomial is given as:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$$

The degree of polynomial is the highest degree of term $x$. Given $m$ pixels represented by coordinates $(x_1, y_1), \ldots, (x_n, y_n)$ for each point in cluster, we will use polynomial regression to find the coefficients $a_n, a_{n-1}, \ldots, a_1, a_0$ of the polynomial with errors:

$$e_i \equiv y_i - p(x_i), \text{ for } i = 1, \ldots, m$$

We are minimizing the sum of squared errors:

$$\sum_{i=1}^{m} e_i^2 \rightarrow \text{min}$$

We can express this equation in matrix form:

$$\vec{y} = p(\vec{x}) + \vec{e}$$

Where $i$-th row of $\vec{x}$ and $\vec{y}$ represents coordinates $x_i, y_i$ of $i$-th cluster pixel. We can expand the form:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{m-1} \\ y_m \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & x_{m-1} & x_{m-1}^2 & \cdots & x_{m-1}^{n-1} & x_{m-1}^n \\ 1 & x_m & x_m^2 & \cdots & x_m^{n-1} & x_m^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \\ a_n \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_{m-1} \\ e_m \end{bmatrix}$$
There are multiple ways of solving this equation while minimizing the sum. The implementation \[32\] used in NumPy library uses pseudo Vandermonde matrix. Detailed principles of this algorithm can be found for example in NASA technical report by Turner, L. R. [50]. The result is the set of the coefficients \(a_n, \ldots, a_1, a_0\).

![ToA](image)

**Figure 3.8:** An example of a cluster fitted by polynomial function of degree 3. Dashed red line is fitted polynomial in the range of cluster.

Simple usage of the cluster points as input for this algorithm without adjustments was proven to be insufficient. For example, cluster in Figure 3.8 is vertically oriented and because we are getting a polynomial as result it is impossible to have two \(y\) values for one \(x\). In the given figure the algorithm would be unable to correctly fit the cluster in the \(x\) range \((25, 40)\) resulting in an incorrect polynomial. To correct this issue we find two most distant pixels in the cluster and using their coordinates \([x_s, y_s]\) and \([x_e, y_e]\) we obtain the angle of the cluster:

\[
\alpha_{\text{cluster}} = \arctan \left( \frac{|x_s - x_e|}{|y_s - y_e|} \right)
\]

As both values in fraction are positive the angle will be in range \((0^\circ, 90^\circ)\). If the \(\alpha_{\text{cluster}}\) exceeds \(45^\circ\), we will invert the \(X\) and \(Y\) axis. The algorithm will then yield appropriate results as shown in Figure 3.8.

The next necessary adjustment is a normalization of the input values. To have a possibility to compare the polynomials we need to transform the \(\vec{x}\) values into the range \((0, 1)\). For that we need to know the minimum and maximum of the coordinates:

\[
x_{\min} = \min (\vec{x}) , \quad x_{\max} = \max (\vec{x}) ,
\]

\[
y_{\min} = \min (\vec{y}) .
\]

Then, for each pixel and its pair of coordinates \((x_i, y_i)\) we compute its new coordinates:

\[
x'_i = \frac{x_i - x_{\min}}{x_{\max} - x_{\min}},
\]

\[
y'_i = \frac{y_i - y_{\min}}{x_{\max} - x_{\min}}.
\]
This transformation ensures comparability of polynomials and preserves ratio of $\vec{x}$ and $\vec{y}$.

Figure 3.9: Random points from a $\sin$ function are selected and then fitted using polynomial regression. For polynomials of rank 5 and 7 almost perfect fits are obtained.

The last parameter to decide is the degree of polynomial that we will try to fit to the cluster. As the winding of clusters rarely has more than two “curves”, we used $\sin(x)$ in range $\langle 0, 2.5\pi \rangle$. In Figure 3.9 there is depicted the original function and the points selected from it. Linear and polynomial regression of degree 2 are unable to properly follow the shape of original function. Polynomial regression with rank 5 and above are almost perfect on the given function and range with minimal error. We have concluded that for our purposes (distinguishing between curly and straight track) polynomial rank 3 is sufficient. In comparison with regression of rank 5 we have two coefficients less and faster computation while keeping the general shape approximation correct. Nevertheless, in the future, more detailed inspection of higher rank regression can bring better results.
3.3.2 Morphology of clusters

Figure 3.10: Complex events can occur in mixed radiation field. This cluster example was measured in ATLAS experiment in CERN. The event is probably the result of an ion collision with the nucleus of the silicone sensor. Nuclear fission results in multiple charged tracks to appear. The collision happened at coordinates [180, 120]. Measuring both Time over Threshold and Time of Arrival simplifies the localization of the event core.

For complex events as shown in Figure 3.10 that are observed in mixed radiation fields, we aim to describe its properties in general. For example, for heavy lead ions we can observe the number of delta electrons. We will focus on individual branches of the cluster as these effects resemble a tree structure (i.e. delta electron, beta decay etc. are creating branches from the path of the original ion).

Before we study the morphology of a cluster, we will remove the halo pixels. The Timepix3 device can effectively measure from 3.6 keV and therefore we can use energy threshold around 4.5 keV (values below 3.6 keV are already in range of noise). Comparative image is shown in Figure 3.11.
Figure 3.11: In the figure, stripping off the halo effect in measurement is shown. Pixels with energies below 4.5 keV are omitted and topology of the cluster is simplified. Left image is original cluster and right is after using the thresholding.

Finding the branches can be done using different approaches, such as Hough transform \[12\] or skeletonization. Given the above mentioned tree structure of the clusters, we used the second method with additional adjustments. Skeletonization emphasizes geometrical properties of a cluster and thus highlights the individual branches. We have chosen the algorithm proposed in the article *A Fast Parallel Algorithm for Thinning Digital Patterns* \[52\] and implemented in the scikit-image library \[33\], for further details, see next Section 3.3.3. After performing the skeletonization we select pixels with only one neighbor as potential ends of branches (i.e. branch vertex). Pixels with three or more neighbors are marked as possible event cores. Figure 3.12 depicts results of the algorithm with marked branch ends and event cores.

Figure 3.12: The image on the left is the original cluster. The right image shows the skeleton of the cluster. Orange circles mark end of branches. The green circles are possible event cores.
In complex events, potential cores are scattered in close proximity. In a similar process as described in Chapter 2.4 we join overlapping cores. Two cores overlap in case of a common neighbor. This yields in most cases one single broad core. To determine individual branch paths, we start in the vertex and add pixels until we reach the event core.

In the future, we can utilize all the known features from the previous subsection to describe separated branches. To distribute pixels from the original cluster between sections (i.e. branches or event cores) we could use euclidian distance to find nearest corresponding section skeleton and therefore we could obtain separated lists of pixels representing branches or event cores.

For example, alpha decay will have high energy with low winding opposed to delta electron that will be curly (i.e. high winding) and will have low energy. An entire exotic cluster is determined by its general characteristics combined with attributes of individual branches and event cores. Completely described cluster is shown in Figure 3.13.

![Figure 3.13: After joining the event cores we can observe one remaining event core (marked as black rectangle with total energy as label). Each branch has its winding depicted as polynomial function.](image)

### 3.3.3 Skeletonization

In this section, we will describe in more details the idea behind the skeletonization algorithm used in the previous section. The method used in the library is based on [A Fast Parallel Algorithm for Thinning Digital Patterns][52] and for more details reading of this article is advised. This algorithm is iterative and runs on a binary image (represented as matrix of boolean values). The idea is to iteratively thin the image in a two step loop with focus on deleting north-west and south-east points. The value of each pixel in iteration \(n + 1\) is based on its value and values of the 8-neighbors in \(n\)-th iteration (this allows parallelization of matrix calculations). Indexing \(P_1 \cdots P_9\) of the \(3 \times 3\) matrix window is important for correct results and is shown in Figure 3.14.
Figure 3.14: Set of pixels representing 3x3 matrix with correct order for skeletonization.

One iteration is a combination of the following two steps:

- We remove point $P_1$ if satisfies the following:
  - $2 \leq \text{number neighbors of } P_1 \leq 6$,
  - Number of patterns 01 in ordered set $P_2 \cdots P_9$ is one,
  - $P_2 \ast P_4 \ast P_6 = 0$ and
  - $P_4 \ast P_6 \ast P_8 = 0$.
  
  If at least one condition is not satisfied, the point $P_1$ is not removed.

- We remove point $P_1$ if satisfies the following:
  - $2 \leq \text{number neighbors of } P_1 \leq 6$ (the same as above),
  - Number of patterns 01 in ordered set $P_2 \cdots P_9$ is one (the same as above),
  - $P_2 \ast P_4 \ast P_6 = 0$ and
  - $P_2 \ast P_6 \ast P_8 = 0$.

We iterate until there is no pixel to remove. The first two conditions ensure that end points are not deleted and the the points lying between the end points are preserved. Given the equations, the first step removes east or south boundary pixel or north-west corner. Similarly, for the second step we remove north or west boundary pixel or south-east corner point. The algorithm is finite and can be parallelized.
4. Classification of clusters

In the previous chapters, we have covered the whole process of capturing ionizing particles with the Timepix3 device, the readout of the data from the chip using the Katherine readout followed by the data transmission to the measurement computer, the clustering of pixels into separated events and the annotation of the clusters (i.e. feature selection). In this chapter, we will focus on the question, how to categorize the measured clusters into the groups laid out in the Chapter 3 Section 3.2.1. Furthermore, we will then apply appropriate clustering methods on the exotic (or unknown) events. Please note, that due to the semantic overload of term clustering, we will consider clustering to have the data analysis meaning (i.e. clustering represents grouping of events based on their properties) in this chapter.

The measured data are commonly from mixed radiation fields (as are the data from ATLAS that we are using). We seek to describe the composition of the field and to find and isolate exotic events that can be potentially subject of further physics analysis. We will show how we obtained labeled data, which classification methods we have utilized and the results. In the end of the chapter we will look more closely on the exotic events. The data we have used are from ATLAS experiment in CERN (described in Chapter 1). The measurement was from ATLAS on during 2018.

4.1 Labeling of the clusters

To gain training data for training classification methods we rely on manual labeling of the clusters as there are limited means to acquire simulated data. We use the analysis software presented in Chapter 3 to label the data. In the first manual labeling run, we gathered the following statistics (we have taken all the clusters from measurement beginning):

Following four tables will use these abbreviations: LG - long gamma, HT - heavy track, HB - heavy blob, ST - straight track, CT - curly track, UN - unknown or exotic.

Table 4.1: Results of labeling of unfiltered data.

<table>
<thead>
<tr>
<th>Total samples</th>
<th>Dot</th>
<th>LG</th>
<th>HT</th>
<th>HB</th>
<th>ST</th>
<th>CT</th>
<th>UN</th>
</tr>
</thead>
<tbody>
<tr>
<td>722</td>
<td>3,7%</td>
<td>89,9%</td>
<td>0,2%</td>
<td>0,4%</td>
<td>2,4%</td>
<td>1,8%</td>
<td>1,6%</td>
</tr>
</tbody>
</table>

As we can see from the Table 4.1 the data are heavily unbalanced towards long gamma clusters. We observe a minimum amount of heavy tracks and blobs and just a little bit more straight and curly tracks. Having heavily unbalanced set towards one category has to be taken into consideration. In our case the long gamma class is predominant with almost 90%. In case we would use standard accuracy metric, our classifier would instantly gain an accuracy of over 90% just
by predicting long gammas for all samples. However, we do have ways how to overcome this issue shown in next section.

4.2 Filtering the measurement data

There are multiple aspects of our classification task that we can adjust to solve the problem of unbalanced data. We will now describe the options we have and we will present our path towards well performing classification method. We will also describe our architecture of cascaded classifiers later on.

4.2.1 Unbalanced dataset adjustments

The goal of these methods is to reduce the bias of the dataset.

Reduce predominant class

By finding classifier with the capability to correctly label the predominant class (in our case long gamma), we can, based on its results, filter the dataset and narrow our scope to the remaining data. In our case, we have concluded that we can filter out dot and long gamma categories before continuing with the rest of the six categories.

Stratification of bigger dataset

By collecting more data, the minor groups will get more samples. If we reach a sufficient number of rare group samples, we can then stratify the data into a balanced set. Stratification is a process of sampling each class with a similar number of samples. In our case, to obtain 1000 non long gamma we would need over 10000 samples manually labeled and thus we will try to avoid this approach.

Generating artificial samples

We can go the other way round, instead of reducing the dataset to balanced set, we can create new samples for minor groups. A straightforward method is random copying of samples from infrequent classes. More advanced methods can create samples from existing examples by interpolation between features of similar samples within one class. Such a method is, for example, Synthetic Minority Over-sampling Technique [42]. As we are searching for exotic events with unknown properties, we will use this method in case of failure of other methods.

Fill categories using manual heuristic

To resolve this imbalance we can use the fact that by simple filtering the clusters by size we can omit most of the long gammas. This means that we increase the number of samples in “rare” dataset.
4.2.2 Quality of the model

Different methods can be used to promote and measure desired results in the classification model.

Weighting the samples

If we penalize the model for misclassifying minor groups, we can force the model training to focus on these small groups. Similarly, we can give weights to samples in accordance to their distribution in the dataset. We will use this technique to minimize the amount of misclassification of the non long gamma events.

Determining the quality of the classifier

To visualize how we predicted each class we can involve the confusion matrix. This matrix allows to see on the diagonal how accurate the prediction of individual classes is. Inherently it also shows the misclassification between classes by the off diagonal entries. We will use the confusion matrices to present quality of our classifiers as shown in Figure 4.3. Ideally we want the matrix to be diagonal, that denotes that all testing data were labeled correctly.

Changing the metrics

We have shown that with basic accuracy metrics (i.e. sum of success rate across classes) we cannot measure the quality of the classifier correctly. Let us now introduce precision, recall and F1-score \[10\] for binary classification tasks, we will later broaden the definition for multi-class problem. Precision is ratio of correctly classified samples called true to the number of all samples. Precision helps to reduce the number of false positives in minor groups.

\[
Precision = \frac{TruePositive}{TruePositive + FalsePositive}
\]

Contrary, the recall metric penalizes the cost of false negatives (i.e. we want to mark all non long gamma as correctly as possible).

\[
Recall = \frac{TruePositive}{TruePositive + FalseNegative}
\]

Combination of precision and recall yields the metric of model accuracy with emphasis on minimizing false positives and false negatives. In our case we want to find almost all non long gamma events while keeping the amount of falsely labeled long gamma low. This combination is called F1-score, where perfect model is when F1-score = 1 and completely wrong model is when F1-score = 1.

\[
F1-score = 2 \times \frac{Precision \times Recall}{Precision + Recall}
\]

Tone of possible expansion of the definition into the multi-class problem is to redefine the TruePositive, FalsePositive and FalseNegative variables. Given classes \(C\) and the number of classes \(n = |C|\), \(n \in \mathbb{N}\), the metric for multi-class classification is the normalized confusion matrix \(M_{n \times n}\). The value at diagonal \(m_{i,i}\), \(i = 1 \ldots n\) represents TruePositive for class \(C_i\). The sum of column without
diagonal element gives the $\text{FalsePositive}$. And the sum of row without diagonal element gives the $\text{FalseNegative}$. Equally:

- $\text{TruePositive}_i = m_{i,i}$
- $\text{FalsePositive}_i = \sum_{j \neq i} m_{i,j}$
- $\text{FalseNegative}_i = \sum_{j \neq i} m_{j,i}$

We can then compute the $F1$-score as an average of individual classes $\text{[34]}$.

$$\text{Precision}_i = \frac{\text{TruePositive}_i}{\text{TruePositive}_i + \text{FalsePositive}_i}$$

$$\text{Recall}_i = \frac{\text{TruePositive}_i}{\text{TruePositive}_i + \text{FalseNegative}_i}$$

$$F1\text{-score}_i = 2 \times \frac{\text{Precision}_i \times \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i}$$

$$F1\text{-score} = \frac{\sum_{i=1}^{n} F1\text{-score}_i}{n}$$

### 4.3 Known groups classification

We can now characterize the classification task and determine suitable classification method. We require separating the cluster into six defined groups plus one unknown category. We would like to transform the classifier for the C++ online framework described in Chapter 2 to enable live filtering. We chose the classification method to be decision trees. This decision is based on the advantages of this classifier described in the next section. Furthermore, we wanted to apply a well known method, that can be easily understood. To support our hypothesis that selected features are capable of sufficiently describing our proposed cluster types, we created manually a testing decision tree for classifying the most distinguishable classes but without long gamma. The model is shown in Figure 4.1.
The dot category is characterized by maximum size of 4 pixels. The long gamma has maximum energy below 100 keV, with low winding and size up to 60 pixels. The cluster size of long gamma is usually below 40 pixels. If the cluster is very rounded and the maximum energy exceeds 100 keV we can assume it to be in the heavy blob category. A straight track is longer than long gamma with almost zero winding and that distinguishes it from a curly track where winding is much more prominent. Heavy tracks have average energy above 100 keV. Unknown or exotic events have mostly more than one branch after the morphological analysis.

4.3.1 Decision trees

The decision tree classifier is a binary tree in which each node represents a binary decision based on a feature value. The leaf nodes represent the assigned classes. Decision trees are suitable for multi-class classification and can perform well with different types of features combined (e.g., boolean, integer and float). In our case they offer two key advantages:

- interpretability – we can observe and understand the process how the model predicts the class and we can represent the model as a tree with simple boolean logic,
- interoperability – the resulting model can be easily used in a different framework such as our real-time clustering software,
- performance – with limited depth we can find prediction in \( N \) comparisons, where \( N \) is equal to the tree depth bolstering the use in real-time applications.

We have shown that we are able to manually create a decision tree. Now we will explain how decision trees are algorithmically build in the library that we used (see scikit \([1]\)).

Given a vector of samples \( S \), each having a vector of features \( f_i, i = 1, ..., j \) (where each feature can be represented by real number, integer or boolean value)
and given vector of labels $L$. Then the decision tree recursively splits the samples from $S$ into groups such that samples with same class are grouped together.

The tree begins with the root node and all samples. Let each node $n$ have a subset of the vector $S$ as set of subsamples $S_n$ in node $n$. In each node, the set $C_n$ of possible splits is created. Let each potential split $c$ be given by the feature and threshold $c = f_i, t_i$ and splits $S_n$ into two sets $Left$ and $Right$. The split $c$ divides samples in the following way:

$$Left(c) = \{(f, l) | f_i \leq t_i\}$$

$$Right(c) = \{(f, l) | f_i > t_i\}$$

To choose the split $c$ from $C_n$ for the node $n$ impurity function $I$ is used. There are three common impurity functions:

- **Gini**

$$I(S_n) = \sum_k p_{nk}(1 - p_{nk})$$

- **Entropy (i.e. Information Gain)**

$$I(S_n) = -\sum_k p_{nk}\log(p_{nk})$$

- **Misclassification**

$$I(S_n) = 1 - \max_k(p_{nk})$$

where $p_{nk}$ is proportion of class $k$ in node $n$.

Current split $k_n$ is selected using minimum of impurities:

$$k_n = \arg\min_c \left[ \frac{|Left(c)|}{|S_n|}I(Left(c)) + \frac{|Right(c)|}{|S_n|}I(Right(c)) \right]$$

Recurse for subsets $Left$ and $Right$ until the maximum depth is reached or the number of samples in the node is 1, or the number of samples is below a minimal threshold.

In the next Figure 4.3 we showcase an example decision tree. The variables $X_0 \ldots X_6$ are the following features in the order: roundness, winding, size, maximum energy, average energy, total energy, number of branches. The value array stores how many samples of individual classes are at the current tree node.
4.3.2 K-fold cross-validation

We will now describe the idea of cross-validation. In-depth details can be found in the following article: *A Gentle Introduction to k-fold Cross-Validation* [3].

K-fold cross validation is a process to measure the quality of a machine learning model. This procedure produces a less biased estimate of the accuracy of the given model. It is specially useful when we are limited by the amount of the ground truth data. Using this method we split the data into groups. The number of groups is given by \( k \) parameters (thus the k-fold). The steps of the algorithm are:

- Randomly shuffle the data.
- Split shuffled data into \( k \) groups.
- Repeat for each of \( k \) groups:
  - Use selected group as validation set, the rest of groups use as training data.
  - Given the training data, fit the model and evaluate it on the validation set. Store the evaluation results and delete the model.
- Combine and average the results of each model.

The setting of parameter \( k \) will affect the results. Most common values are 5 or 10. Leave-one-out cross-validation is the type of cross validation where \( k = n \) and
$n$ is the number of samples. In this case, we use each sample as a test subject. We can also set $k$ dynamically based on the distribution of our dataset to have groups large enough to statistically represent each class.

### 4.3.3 Step-by-step filtering

As the dataset has a predominant class the long gamma, we chose to use the cascade approach for classification of the data. At each step of the cascade, the goal is to filter out the predominant non exotic particles and gradually gain smaller datasets with more interesting events. We use the incremental steps to permit a subsequent buildup on these algorithms with different focus. We start by creating precise classifier for the largest and the simplest classes (i.e. long gammas and dots). The model is required to:

- split the data into three categories, *dot*, *long gamma* and *other* and
- minimize the false negative error of *other* class.

By cutting out the first two categories we will be left with just 8% of the original data. The minimization of the false negative error will ensure that all (or at least the maximum possible amount) of remaining classes, including exotic events, will remain in the dataset. Otherwise we are at a risk of omitting important events.

From the methods to tackle biased datasets shown in the previous section we will involve using weight for samples and the $F1$-score metric. From the labeled dataset, we separate around 25% of samples as a testing dataset. We will also use 5-fold cross-validation as described in previous section.

As we want to keep the possibility of fast classification in real-time, we will try to reduce the features used to the simpler of them. For example, computation of skeleton and search for branches is a much more complex task compared to search for average energy of cluster. Thus we ruled out the *number of branches* features.

As for the tuning of the decision tree classifier, we have a small labeled dataset and this allows us to use grid search. Grid search $[2]$ is a utility to tune parameters of the given classification method. It uses hyper parameters to search for the optimal group of parameters. It is equivalent to testing each combination of parameters individually. In addition, we perform cross-validation for each generated model. We will now describe specifically which values we use for our training.

We include both *gini impurity* and *information gain* loss functions as hyper parameter. The tree depth hyper parameter is set to be in range from 3 to 15. The minimum leaf size (i.e. amount of samples in leaf node) is in the range from 2 up to 9. To suppress the bias of the dataset, we set weight of the *other* class to be $10 \times$ higher than the weight of *long gamma*. The ratio between the classes is $10 : 1$ so the weight increase should match the distribution. As the scoring function of the hyper parameter search algorithm $F1$-score is used. With the simple accuracy metric we have not met the goals set above and the false negative error was to high as can be seen in Figure 4.3. To determine the quality of the model we use confusion matrices where we want to maximize the bottom right value ($Other \times Other$) and we keep the false labeling of *other* near zero ($Other \times Dot$ and $Other \times LG$). Additionally but not necessary, we try to keep the false positive numbers low too ($LG \times Other$ and $LG \times Other$).
Figure 4.3: Confusion matrix of decision tree with suboptimal results. The false negative number of other class is too high.

With the correct metric the hyper parameter optimizer showed the best result to be:

- criterion – information gain,
- maximum tree depth – 3,
- minimum samples in leaf – 6.

The confusion matrix confirms good results. The accuracy of picking out non long gamma clusters is almost perfect. Although we do not have ideal results, the other two classes, we are able to filter out the 10% of more interesting classes.

Figure 4.4: Model has with high precision classified most of the long gamma events.

We can now save the model and apply the filtering to the dataset. In the resulting filtered unlabeled dataset we can still see some long gamma samples (the manually labeled events did not covered the whole category sufficiently), these can be labeled again. Thus we have labeled another set of data, but now from filtered file. We obtained the following distribution:
Table 4.2: Results of labeling of clusters filtered from raw data using previous classifier.

<table>
<thead>
<tr>
<th>Total samples</th>
<th>Dot</th>
<th>LG</th>
<th>HT</th>
<th>HB</th>
<th>ST</th>
<th>CT</th>
<th>UN</th>
</tr>
</thead>
<tbody>
<tr>
<td>239</td>
<td>0%</td>
<td>57%</td>
<td>7.5%</td>
<td>5.8%</td>
<td>2.5%</td>
<td>14.2%</td>
<td>13%</td>
</tr>
</tbody>
</table>

We train another classifier for the filtered data, that we have labeled. Because we have not found any dots during the labeling of the filtered dataset, we now perform only binary classification.

Figure 4.5: Binary classification between long gamma and other tracks. After first filtering, the data are almost balanced (57% and 43%).

The resulting model has good results with over 90% accuracy for other category. The resulting parameters were:

- criterion – gini-impurity,
- maximum tree depth – 3,
- minimum samples in leaf – 18.

From the second time filtered dataset we now label the remaining four categories curly tracks, straight tracks, heavy blobs and heavy tracks plus one extra category of unknown events (i.e. events of collision, decay etc.). We have also added the number of branches features.

We again use the decision trees with grid search, but now we broaden the search space with increased maximum depth of the tree. The resulting parameters for this classification are:

- criterion – gini-impurity,
- maximum tree depth – 7,
- minimum samples in leaf – 2.
The resulting confusion matrix is shown in Figure 4.6.

Figure 4.6: Classification of double filtered data. The exotic events are separated with high precision, however there are some false positive occurrences between exotic class and heavy track class.

The uncertainty between straight track and curly track is probably due to lack of samples and because of small difference between these two categories. By labeling additional packs of events we can increase the accuracy within these categories.

4.3.4 Classifying all six categories

As a second experimental approach, we chose to use the manual heuristics (without the set that represents real distribution) to gain the balanced set. Using the analysis software we browse and label only clusters with size bigger than 60 pixels. That yielded the following results:

Table 4.3: Results of labeling of clusters bigger than 60 pixels.

<table>
<thead>
<tr>
<th>Total samples</th>
<th>Dot</th>
<th>LG</th>
<th>HT</th>
<th>HB</th>
<th>ST</th>
<th>CT</th>
<th>UN</th>
</tr>
</thead>
<tbody>
<tr>
<td>455</td>
<td>0%</td>
<td>0%</td>
<td>34.7%</td>
<td>24.2%</td>
<td>7.9%</td>
<td>6.8%</td>
<td>26.4%</td>
</tr>
</tbody>
</table>

However we still suffer from an insufficient amount of dot clusters. Dots are by definition only clusters with pixel size from 1 to 4. By labeling another small amount of clusters using maximum size filter of 5 we get enough labeled dots.
Table 4.4: Results of labeling of smallest clusters (i.e. at most than 5 pixels).

<table>
<thead>
<tr>
<th>Total samples</th>
<th>Dot</th>
<th>LG</th>
<th>HT</th>
<th>HB</th>
<th>ST</th>
<th>CT</th>
<th>UN</th>
</tr>
</thead>
<tbody>
<tr>
<td>230</td>
<td>78.6%</td>
<td>21.4%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

By combining the above mentioned results we are able to reduce the percentage of long gammas in the dataset from almost 90% down to 49%.

Table 4.5: Statistics of labeled set. Combined from all previous labelings.

<table>
<thead>
<tr>
<th>Total samples</th>
<th>Dot</th>
<th>LG</th>
<th>HT</th>
<th>HB</th>
<th>ST</th>
<th>CT</th>
<th>UN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1407</td>
<td>14.8%</td>
<td>49.5%</td>
<td>11.4%</td>
<td>8.0%</td>
<td>3.7%</td>
<td>3.2%</td>
<td>9.4%</td>
</tr>
</tbody>
</table>

In this way we have obtained a more balanced training set.
As a testing set we used the data with real distribution. Given these sets we now learn the classification model of decision tree.

Table 4.6: Statistics of manual heuristic set.

<table>
<thead>
<tr>
<th>Total samples</th>
<th>Dot</th>
<th>LG</th>
<th>HT</th>
<th>HB</th>
<th>ST</th>
<th>CT</th>
<th>UN</th>
</tr>
</thead>
<tbody>
<tr>
<td>685</td>
<td>26%</td>
<td>7%</td>
<td>23%</td>
<td>16%</td>
<td>5%</td>
<td>5%</td>
<td>18%</td>
</tr>
</tbody>
</table>

We will again use grid search for the best parameters, however now without \textit{F1-score} (as the dataset is more balanced and we are not filtering the data). We involve all features described in previous chapter. The parameters we obtained were:

- criterion – \textit{information gain},
- maximum tree depth – 5,
- minimum samples in leaf – 2.
Figure 4.7: Results of classification directly into six categories using manual heuristics.

From the confusion matrix we can see that we have obtained very good results. However, these results are strongly based on our previous heuristics. In case we would be unable to guess correctly how to manually get data to balance the dataset, we would be unable to classify correctly. Another drawback of this setting is that it is tailored for a specific dataset and categories and in case of different measurement we would be forced to start from a scratch. Having the classifiers in cascade allows to reuse certain parts of the cascade it is more robust for wider amount of measurements. This experiment was done to showcase the possibility to quickly create a sufficient classifier in case we are sure about our manual heuristic.

4.4 Conclusion of classification

After labeling the raw data from an experiment we found out that the data is heavily unbalanced. During the future experiments, the chances of a biased dataset is high. To tackle this, we presented solutions to classify such type of data. From these methods, we explored different metrics and created cascade of algorithms – label, fit model, filter, label, fit model, filter and finally classify. We have shown that our engineered features are capable of describing the predefined classes using decision trees. However, we have also shown that with good manual heuristic, for specific dataset, we can create classifier without the need of the above cascade. By choosing decision trees we are able to use this classifier in our C++ framework without affecting the performance. In the future, the remaining unknown category can be further divided and clustered. The different classifiers can be combined with the current solution. Tested models can be then rewritten into C++ and applied in real time data acquisition.
Conclusion

The goal of the thesis was to develop a framework for low level data processing of data taken with hybrid pixel detector Timepix3 and to perform data analysis and classification on the processed and measured data with focus on search for exotic events in real-time. All of the set goals were achieved.

Physics and electronics background

In the first chapter we introduced the reader into the topic of particle detection. We have described the technical details of Timepix3 chip and how the detection of ionizing particles works. Furthermore, we explained details of the Katherine readout that, together with Timepix3, allows to measure the radiation fields.

Low level data framework

Based on these two devices, we build up our software for low level data processing. We have created a versatile framework with multiple working use cases. Our framework supports both Graphical User Interface (GUI) and Command Line Interface (CLI) to suit both end-user operation or remote server application. We have created and tested means to communicate with a Katherine device over UDP. This tool includes modules suitable for monitoring ongoing measurement (e.g., live statistics and live display of the sensor). Also supports post-measurement offline data processing. The low level data processing consist of (beside other things) non-trivial grouping of pixels into separate events even in high pixel data rates. The software was developed in modern C++ with focus on performance and with modular architecture (also with support for live event filter plug-ins). All features were tested on the major platforms (macOS, Windows 10 and Debian based Linux). Results of this work were presented at the conference Connecting the Dots / Intelligent Trackers 2019, Valencia, Spain [7] and are followed by a proceedings article.

Characterization and classification of events

In the last section we characterized the events. We set up basic features describing the clusters. Furthermore, we applied advanced techniques to describe the morphology of individual events. To visualize the events and test our feature descriptors, we create a second tool, this time written in Python to have the possibility to use the broad machine learning and image recognition libraries (e.g., scikit [35]).

As we were in a field without existing labeled data, we used our second tool to manually label the events. With these created datasets we identified possible ways how to classify the events into six categories. Two approaches were shown on how the classification can be performed. We tested our resulting classifiers on real data with good results and achieved a reduction of large datasets into a fraction of its previous size by finding exotic events. This can be also used as a guide how to create classifiers for different radiation fields.

The results of this thesis can be directly used in:
• physical analysis for offline clustering of data,
• visualization during experimental measurements and
• filtering of measured data.

Some parts of the developed software are currently used by physicians for data preprocessing.

Further research

Due to the modular design of the developed framework, it can easily be extended to work with future developments of the Medipix collaborations (namely Timepix2 and Timepix4). The performance of the code in very high flux environments has not yet been sufficiently tested. Future efforts could be dedicated to speed up the clustering.

In the subject of machine learning and classification improved algorithms could be developed by increasing properly labelled data, either through improved detector description in physics simulation or by gaining training data in well defined particle fields.

Final notes

The work was supported from European Regional Development Fund-Projects:

"Van de Graaff Accelerator - a Tunable Source of Monoenergetic Neutrons and Light Ions”
(No. CZ.02.1.01/0.0/0.0/16_013/0001785) and

"Engineering applications of microworld physics”
(No. CZ.02.1.01/0.0/0.0/16_019/0000766).

Publication contribution:

Transition radiation measurements with a Si and a GaAs pixel sensor on a Timepix3 chip [43].
Bibliography


## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Timepix at the International Space Station (In the right picture inside the yellow circle). On the left is a detail of Timepix sensor with USB interface [27]. The Timepix chip was sent to ISS by members of the Medipix collaboration at the Institute of Experimental and Applied Physics in Prague and the University of Houston, Texas. Its task is to monitor the radiation environment for ionizing dose assessment inside the space station [26].</td>
</tr>
<tr>
<td>2</td>
<td>Example of particle tracks observed within a one second integration window. It is similar to a picture from standard CMOS camera, but with a resolution of $256 \times 256$ pixels. The color depicts the energy deposited in a pixel. Details are explained later in the work.</td>
</tr>
<tr>
<td>1.1</td>
<td>Cloud chamber picture of tracks of ionizing radiation [6].</td>
</tr>
<tr>
<td>1.2</td>
<td>Annotated model of a hybrid pixel detector [47] .</td>
</tr>
<tr>
<td>1.3</td>
<td>Schematics of a clock in the Timepix3 chip during ToA and ToT mode [48].</td>
</tr>
<tr>
<td>1.4</td>
<td>Katherine readout device without Timepix3 chip. Backside (shown in figure) has RJ-45 Ethernet port, GPIO port and DC input. In front VHDCI (Very-High-Density-Cable-Interconnect) 68-pin connector for the Timepix3 is present and LEMO connector for bias.</td>
</tr>
<tr>
<td>1.5</td>
<td>Timepix3 device (in the center of figure) in the Large Hadron Collider in CERN. Due to high radiation, the Katherine readout electronics were placed over 100 m away from the chip in a cavern. Timepix3 measures the radiation field in the tunnel continuously.</td>
</tr>
<tr>
<td>1.6</td>
<td>Approximated sensor response to the deposited energy using multiple sources with known energy [39].</td>
</tr>
<tr>
<td>1.7</td>
<td>Time of arrival inaccuracy. Although all events have the same peak time, the calculated ToA is different. This effect can occur when four neighboring pixels are hit at once [49].</td>
</tr>
<tr>
<td>1.8</td>
<td>Positions of the Timepix3 devices in the ATLAS experiment [40].</td>
</tr>
<tr>
<td>2.1</td>
<td>Complete data flow from the Timepix3 chip to data analysis. White steps represent lowest hardware processing which is described in Chapter 1. Green steps are discussed in this chapter and the yellow ones are described in the following Chapter 3 and 4. Red boxes distinguish different software packages.</td>
</tr>
<tr>
<td>2.2</td>
<td>Pipeline of data processing. Data flow from the top to the bottom. Replaceable modules are at the levels of Feeders, Readers and Outputters. MTQueue denotes a multithreading safe queue for data transmission between the threads.</td>
</tr>
<tr>
<td>2.3</td>
<td>Lifecycle of components in framework.</td>
</tr>
<tr>
<td>2.4</td>
<td>The offset datagram is used to transmit the most significant part of the final time of arrival. The offset is an integer value.</td>
</tr>
<tr>
<td>2.5</td>
<td>Data-driven datagram for one pixel hit. Contains $x$ and $y$ pixel matrix index, timestamps and time over threshold of the pixel.</td>
</tr>
</tbody>
</table>
2.6 Cluster versus integration. Source of data is ATLAS experiment. Black background in (a) represents zero energy pixels and the whole matrix \( M_{256,256} \) is displayed. For clear view, in (b) zero energy pixels are white and all non-zero energy pixels are represented using the color scale as in (c) and the image is cropped to matrix \( M_{80,80} \).

2.7 Type of pixel neighbors. Yellow pixels denote neighbors of the red center one.

2.8 Examples of quadtrees.

2.9 Graph representing the average time to cluster file with given leaf size.

2.10 Filter widget for size and energy clustering. The widget is a part of the output, see the following sections. The range of the values is from zero to the maximum value during measurement. By moving the right slider to the end we can set the maximum value to infinity.

2.11 Example of loaded filter plugins.

2.12 Event statistics output. In the left column there are controls of individual graphs. In the middle part, there is a space for up to three filters. In the right section we can see the produced graphs. Each filter represents one colored line in the charts. The size and energy filters have the color displayed in the control panel. The yellow color is representing the basic class filter. The settings can be hidden with the top left “hamburger button”. Replayed data are from a measurement at Super Proton Synchrotron experiment in CERN.

2.13 Integration window. With the first column we can adjust the time window and refresh time of the window. With the refresh time bigger than time window, we will possibly skip (not see) some of the clusters. Middle column is used for up to three filters which are substractive (cluster must satisfy all of the filters). Right part of the visualization shows ToA and ToT canvases. Color bars and their colormaps are automatically adjusted to cover whole range of values in canvases. The settings can be hidden.

2.14 Comparison of the 100 seconds long measurement with and without filtering involved. On the left we kept all of the clusters captured during the integration time. On the right we can see only clusters with the size bigger than 40 pixels.

2.15 User interface for saving the clustered data. In the left part, we can choose directory to save the results (if it is empty, the current directory of the program is used). To avoid overriding files, the filename is automatically generated based on the current time. In the second part there are possible filters. The user can store filtered and unfiltered data simultaneously (two files are created).
3.1 Complete data flow from the Timepix3 chip to data analysis. White steps represent lowest hardware processing which is described in Chapter 1. Green steps are discussed in this and following chapters and the yellow ones are described in the previous Chapter 2. Red boxes distinguish different software packages.

3.2 The architecture of the Data Browser. The clustered data are read from a CSV file and parsed using the CSV File Reader. The Data Browser controls the file reader (e.g., read next, seek to index, read previous) and passes clusters to further processing. The ToA and ToT graphs are displaying the current cluster with colors representing energy and difference from the first time of arrival in cluster respectively. The 3D ToT graphs allows optional 3D visualization where the third dimension is the energy in keV. The experiment visualization component is meant for feature testing and has its own canvas.

3.3 Visualization of the cluster using Matplotlib. On the left side there is a cluster with color representing the relative time of arrival. The other image shows the energy deposition in keV at individual pixels. Image is zoomed into the part of the 256 × 256 pixels chip where the cluster appeared.

3.4 3D visualization of the energy of a cluster.

3.5 Six categories of known cluster types.

3.6 Exotic event representing decay of a particle.

3.7 An example of computing the convex hull on three functions. From left to the right: \( \sin(\pi \times 0.8x) \), \( \sin(\pi \times 2x) \) and \( \sin(\pi \times x) \). The area comparison is: 89.4%, 100% and 89.5%. Although we can immediately see the difference between the winding of individual functions, we are unable to correctly distinguish them using the convex hull area.

3.8 An example of a cluster fitted by polynomial function of degree 3. Dashed red line is fitted polynomial in the range of cluster.

3.9 Random points from a \( \sin \) function are selected and then fitted using polynomial regression. For polynomials of rank 5 and 7 almost perfect fits are obtained.

3.10 Complex events can occur in mixed radiation field. This cluster example was measured in ATLAS experiment in CERN. The event is probably the result of an ion collision with the nucleus of the silicone sensor. Nuclear fission results in multiple charged tracks to appear. The collision happened at coordinates [180, 120]. Measuring both Time over Threshold and Time of Arrival simplifies the localization of the event core.

3.11 In the figure, stripping off the halo effect in measurement is shown. Pixels with energies below 4.5 keV are omitted and topology of the cluster is simplified. Left image is original cluster and right is after using the thresholding.

3.12 The image on the left is the original cluster. The right image shows the skeleton of the cluster. Orange circles mark end of branches. The green circles are possible event cores.
3.13 After joining the event cores we can observe one remaining event core (marked as black rectangle with total energy as label). Each branch has its winding depicted as polynomial function. 51

3.14 Set of pixels representing 3x3 matrix with correct order for skeletonization. 52

4.1 Manually created decision tree. Similar tree is used in the demo filtering plugin in Chapter[2]. 57

4.2 Example of tree generated by decision tree classifier. Tree depth was limited to 3. 59

4.3 Confusion matrix of decision tree with suboptimal results. The false negative number of other class is too high. 61

4.4 Model has with high precision classified most of the long gamma events. 61

4.5 Binary classification between long gamma and other tracks. After first filtering, the data are almost balanced (57% and 43%). 62

4.6 Classification of double filtered data. The exotic events are separated with high precision, however there are some false positive occurrences between exotic class and heavy track class. 63

4.7 Results of classification directly into six categories using manual heuristics. 65
List of Tables

1.1 Timepix3 general properties .......................... 7
1.2 Connection options between Katherine readout and Timepix3 chip. 9
4.1 Results of labeling of unfiltered data ...................... 53
4.2 Results of labeling of clusters filtered from raw data using previous classifier .................. 62
4.3 Results of labeling of clusters bigger than 60 pixels .......................... 63
4.4 Results of labeling of smallest clusters (i.e. at most than 5 pixels) .................. 64
4.5 Statistics of labeled set. Combined from all previous labelings .................. 64
4.6 Statistics of manual heuristic set .......................... 64
A. Attachments

The attachments in a digital archive are the following:

Clusterer framework - user manual - online GUI version
The manual describes how to operate the version of Clusterer, that is capable of direct measurement from Katherine device.

Clusterer framework - user manual - offline GUI version
Detailed description of Clusterer framework with details about individual extensions in the software are described. All visualization modules are shown and their control explained.

Clusterer framework - user manual - offline CLI version
Quick introduction into command line version of Cluster, where command line arguments are explained.

Clusterer framework - programmers manual
Technical details about Clusterer are explained to simplify future development of the software. Build and run instructions are included. Architecture of software is shown, key parts are briefly described and lifecycle of objects is discussed.

Analyzing PyQt - user manual
Control of the utility is explained and manual labeling is explained.

Analyzing PyQt - programmers manual
Summary of dependencies of this utility and introduction to important parts and scripts.