



**FACULTY  
OF MATHEMATICS  
AND PHYSICS**  
Charles University

## **BACHELOR THESIS**

Tomáš Čelko

### **Support for annotating and classifying particles detected by Timepix3**

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: RNDr. František Mráz, CSc.

Study programme: Computer Science

Specialization: IOI

Prague 2021

I declare that I carried out this bachelor 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 Coll., 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 paragraph 1 of the Copyright Act.

In..... date.....

signature

First and foremost, I would like to express my sincere gratitude towards my thesis advisor RNDr. František Mráz, CSc. from the Faculty of Mathematics and Physics, Charles University, for his help and guidance during the work on the thesis.

I would like to thank MSc. Benedikt Ludwig Bergmann, PhD. from the Institute of Experimental and Applied Physics, Czech Technical University, for his support.

I would also like to express my gratitude towards Declan Garvey for his suggestions and sharing the essential data which I widely used in the thesis.

Another person I would like to thank is Mgr. Lukáš Meduna from the Institute of Experimental and Applied Physics, Czech Technical University, for the introduction into the problem, as well as for creating the application upon which I could build my thesis.

I am also grateful to my family for their unending support. And last but not least, I would like to offer my special thanks to my girlfriend Ivana Piačková for the beta-testing of the software and her encouragement during the writing of the thesis.

**Title: Support for annotating and classifying particles detected by Timepix3**

**Author:** Tomáš Čelko

**Department / Institute:** Department of Software and Computer Science Education

**Supervisor of the master thesis:** RNDr. František Mráz, CSc., Faculty of

**Mathematics and Physics, Charles University**

**Abstract:**

Hybrid pixel detectors like TimePix3 can capture gigabytes of data on various particles in a second. However, in such measurements, a vast majority of these particles represent already well-known particles. Distinguishing between the types of particles is the first step in searching for extraordinary particles. It is a non-trivial task often done by physicists.

Source data consists of clusters that are groups of pixels of the detector hit by a particle or its secondary particles when the particle decays. Manual processing of the data to such an extent is inefficient. We created a set of tools for visualizing clusters, computing properties of clusters, filtering clusters based on their properties, and training neural network classifiers. Trained classifiers can be merged into a tree structure, offering a better utilization of unevenly distributed types of clusters.

Based on simulated labeled data, we trained multiple classifiers and evaluated their performance on the test dataset of clusters.

**Keywords:** cluster, classification, Timepix3, neural networks

# Contents

Introduction .....	1
1 Analysis.....	3
1.1 The Medipix family and the TimePix3 detector .....	3
1.2 Input format and calibration .....	5
1.3 Cluster visualization .....	7
1.4 Classification .....	9
2 Goals of the thesis .....	14
3 Cluster processor applications.....	16
3.1 Specification .....	16
3.2 A little about neighboring.....	17
3.3 Calculating the features .....	18
3.4 Cluster Viewer.....	21
3.5 Cluster Filter .....	31
3.6 Description Generator .....	32
3.7 Classifier applications .....	36
4 Development documentation .....	42
4.1 Cluster Viewer.....	43
4.2 Filter and Description Generator .....	44
4.3 Classifier applications .....	47
5 Experiments.....	50
5.1 Classifier Parametrization .....	52
5.2 K-Fold cross-validation of single-layered models .....	55
5.3 Multi-layered classifier.....	56
6 Conclusion .....	60

# Introduction

In nuclear physics, there have been many efforts to visualize and detect elementary particles. For this purpose, various detectors were invented. Several such detectors are now members of the so-called Medipix detector family [1]. The most recent member of the family is called the TimePix3 detector. In terms of elementary particle detection, TimePix3 achieves state-of-art performance. During a specified timeframe, the detector captures a set of particle trajectories as hits in a  $256 \times 256$  matrix of pixels. However, in some cases, a single particle emits secondary particles that can interact. Instead of analyzing pixel hits one by one, we analyze groups of hits corresponding to its secondary particles – so-called clusters.

So far, there have not been many publications dealing with the processing, filtering, visualization, or classification of these clusters. In the filtering task, one needs to make sure the algorithm is fast because the size of the cluster dataset captured by TimePix3 detector can reach gigabytes of data over a short timeframe. The classification process can be challenging because the trajectory of the cluster depends on the angle at which the particle enters the field of the detector. Furthermore, the distribution of various types of clusters in the standard observation is usually very uneven. For instance, most of the data received from detectors like ATLAS [2] contains only simple traces consisting of a few pixels and does not provide much information for the analysis. This fact causes problems for many machine-learning-based approaches because the fraction of the rare and more complicated types of clusters is often very small.

## Goals of the thesis

Our main goal is to create a set of processing tools that would enable physicists to analyze the clusters and their properties. The input will be a collection of clusters from the Clusterer application [3]. Firstly we need to provide support for filtering the clusters based on their attributes. Secondly, we visualize them individually so that the users can see the cluster as a 2D and 3D image. Because the number of clusters in some datasets can be overwhelming, another goal is to make a tool that calculates the properties for the whole collection of clusters. The calculated

properties of the clusters enable us to design a neural network-based classifier capable of classifying various clusters, which is our final goal. Eventually, the developed classifier could also detect extraordinary clusters, displaying exotic or even unseen particles.

## **Thesis layout**

First, in Chapter 1, we will introduce TimePix3 detector as a member of the Medipix detector family. Then, we will go over the data format and define various features of a cluster and the methods of their calculation. After we are familiar with the features, we discuss the classification task and its difficulties. When we are done with the analysis part, we specify our goals in a short Chapter 2. Based on these goals, we implement a solution which is in detail described in Chapter 3. This chapter contains elementary information about all of the applications that were created. A more-detailed description of our solution can be found in Chapter 4. Because the classification process can be considered non-deterministic, we will provide a brief statistical analysis of the results in Chapter 5. In concluding Chapter 6 we summarize the outcomes of the thesis and propose direction how the developed tools could be further extended.

# 1 Analysis

In this chapter, we discuss the detection of elementary particles with hybrid pixel detectors, as well as the tools and methods suitable for the analysis of the detector output. In Section 1.1, we introduce the Medipix detector family. Section 1.2 presents the format of the data from the Timepix3 detector. Then, in Section 1.3 we move to the visualization of the Timepix3 data, and last but not least, in Section 1.4, we discuss the classification methods for the groups of elementary particles called clusters.

## 1.1 The Medipix family and the Timepix3 detector

It was the 1990's when some of the researchers from CERN (European Council for Nuclear Research, in French *Conseil Européen pour la Recherche Nucléaire*) came with an idea to transfer the devices, primarily developed for experiments in the Large Hadron Collider (LHC) beyond particle physics. The first collaboration with such a goal started under the name Medipix1. It was the collaboration between the University of Freiburg, University of Glasgow, and Napoli and Pisa Universities together and CERN. So far, there have been four Medipix collaborations, each with its own specific goals. The chips developed in these collaborations are known as the members of the Medipix detector family.

The first member of the family was the Medipix1 chip (1997) [4], consisting of  $64 \times 64$  pixels acting similarly to a digital camera – counting the hits of elementary particles while the shutter is open. A few years later, the Medipix2 chip was developed, leading to the first Timepix chip (2006) [5] being invented. This was the first chip that can be programmed to record one of the following properties:

- **Particle hit count:** The number of hits detected by each pixel in one tick of the internal clock. (similar to Medipix1)
- **Time over the threshold:** Each pixel is assigned an energy threshold level (THL). When a charged particle approaches the pixel, the energy captured by the pixel rises. The time interval during which the energy remains above the threshold we call the *time over the threshold (ToT)*.



This attribute is often measured as the number of ticks of a detector clock.

- **Time of the arrival:** The absolute time since the start of measurement until the energy threshold level is reached we call the *time of arrival (ToA)*. Both time over the threshold and time of the arrival are visualized in the plot in Figure 1.1.

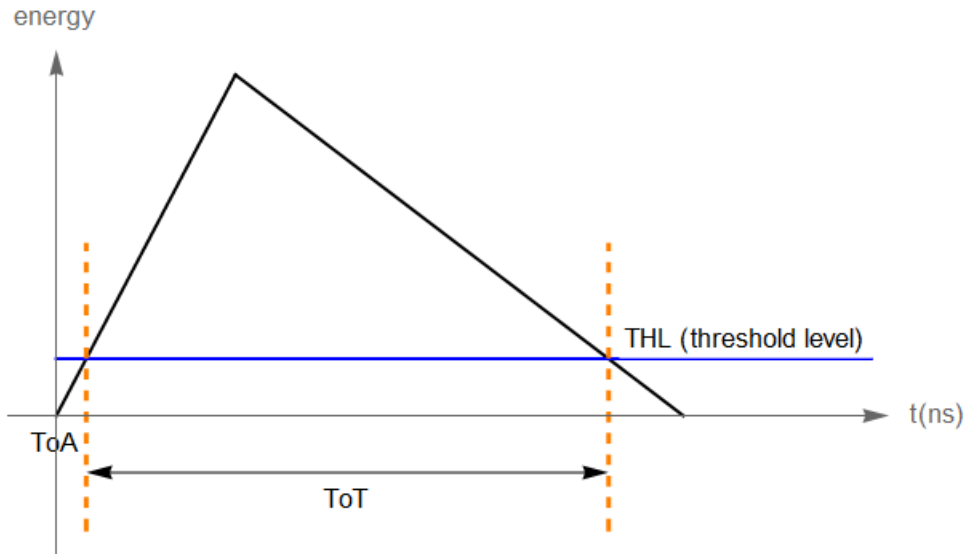


Figure 1.1 Time of arrival and time over the threshold visualization, graph depicts dependency of energy on time

In 2013 the new Timepix3 chip was introduced to the family:

*"Timepix3 is a general-purpose integrated circuit suitable for readout of both semiconductor detectors and gas-filled detectors. Compared to its predecessor Timepix the circuit has more functionality, better time resolution, and more advanced architecture for continuous sparse data readout with zero-suppression."*  
[1]

Zero-suppression means that there is no data output from the detector unless non-zero energy input is captured. This allows better efficiency in data collecting and storage but also in data analysis. The device utilizes a  $256 \times 256$  pixel matrix where the size of each pixel is  $55\mu\text{m}$ , achieving a time resolution of  $1.56\text{ns}$ . The Timepix3 chip is nowadays used in the CERN LHC to detect sets of elementary particles, so-

called clusters. A cluster is formed by traces of particles that either interacted with each other or one of the particles emitted the other.

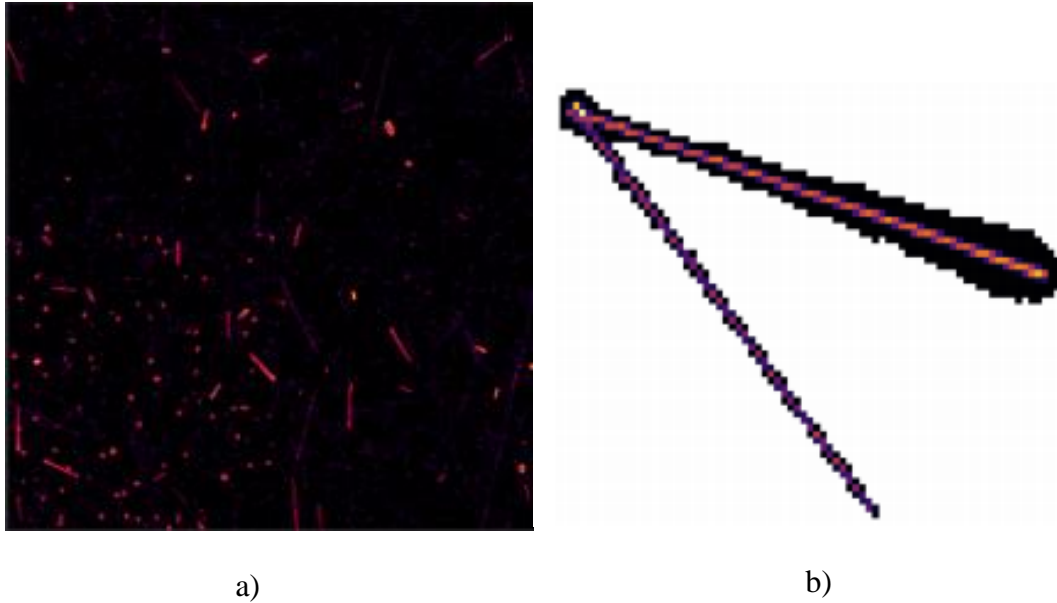


Figure 1.2 Trajectories detected by Timepix3 over 1 second (a) and a single cluster consisting of two particle trajectories (b) [9]

## 1.2 Input format and calibration

Because the output of the Timepix3 chip over some timeframe can contain multiple clusters, Lukáš Meduna developed the Clusterer application [3] to process the raw input from the detector and separate the clusters from each other. This means that instead of processing the data in the raw format, we use the data in the MM (clustered) format. The MM data format consists of three text files – with extensions `.ini`, `.cl`, and `.px`.

- `Ini` file contains the name of the measurement and the names of the `cl` and `px` files associated with the given measurement. By default, this file expects to find the `cl` and `px` file in the same directory where the `ini` file is located.
- `Cl` file consists of the primary data about the cluster collection, where each line represents a single cluster. A line contains the first time of arrival of the cluster in ns (since the start of the measurement). This is

followed by the size of a cluster and the number of the line in the `px` file (and the number of the byte) where the cluster starts.

- `Px` file includes data about each pixel in the cluster, namely  $x$  and  $y$  coordinates of the pixel, the time of arrival in ns, and the energy in keV.

File	Format	Example
<code>ini</code>	[Measurement (or any string ending with a newline char)] PxFile=[Relative path from the parent directory of .ini to px file] ClFile=[Relative path from the parent directory of .ini to cl file]	Measurement 123 PxFile= Clusters_px.txt ClFile= Clusters_cl.txt
<code>cl</code>	[First ToA (decimal)] [Pixel Hit Count ( integer 0-2 <sup>32</sup> )] [LineOfStart in px file ( integer 0-2 <sup>32</sup> )] [Byte of start in px file (integer 0-2 <sup>32</sup> )]	12345.647 100 5 30
<code>px</code>	[x coordinate of the pixel ( integer 0-255)] [y coordinate of the pixel (integer 0-255)] [ToA (decimal)] [Energy (decimal)]	123 128 15540 14.235

**Table 1.1 Input MM file format**

There are two kinds of MM formats, which only differ in the `px` file:

- **Calibrated** data have the structure as displayed in Table 1.1.
- **Non-calibrated** data are very similar to the calibrated ones, but instead of the energy attribute in `px` file, they use the *ToT* attribute.

To determine whether a file is calibrated or non-calibrated, we can open the `px` file and look for the last column. If the decimal part of the values is zero in every pixel, we know we are dealing with non-calibrated data because the *ToT* is measured as a number of ticks of the detector, which is an integral value. However, if the values have a non-trivial decimal part that indicates the data is already calibrated.

The calibration is the process of replacing the *ToT* attribute with the corresponding energy. The energy deposited in the pixel is a function of the *ToT* but also four other parameters of a pixel, denoted by the letters  $a$ ,  $b$ ,  $c$ , and  $t$ . These parameters for each pixel are set up during a calibrating measurement and are usually stored in separate text files as a 256×256 matrix of decimal numbers. It holds:

$$ToT = aE + b - \frac{c}{E - t}$$

In the equation,  $E$  represents the energy deposited in the pixel, with  $a$ ,  $b$ ,  $c$ , and  $t$  being the calibration parameters.

### 1.3 Cluster visualization

The clusters we obtain from the Clusterer [3] can be represented as a set of pixels, each with its own value of  $ToA$  and energy. These pixels can then be visualized on a 256x256 bitmap, where a pixel is assigned a color according to the value of the pixels attribute. In this subchapter, we introduce some methods for further cluster analysis based on its 2D image. These methods include skeletonization and z-coordinate calculation.

#### *Skeletonization*

Skeletonization of a binary image is defined as the thinning process that outputs a simpler version of the original image, the so-called skeleton. An essential requirement for skeletonization is to have the image in binary format – each pixel is either white or black. Because the image of a cluster is not binary, we assign the value 1 to each pixel with non-zero energy and a value 0 to a pixel with zero energy. The skeleton should preserve the original shape of an image. In fact, the definition of skeletonization is not exact, which means various approaches can be used, each possibly outputting a unique skeleton, as displayed in Figure 1.3. The skeletonization process could be helpful when analyzing the clusters because of these advantages:

- The skeleton of the cluster preserves the key information about its shape.
- Skeleton reduces the number of pixels of the image, making it more compact in terms of memory consumption.
- Skeleton enables us to view the image as a branched curve in the plane, which could be helpful for further analysis.

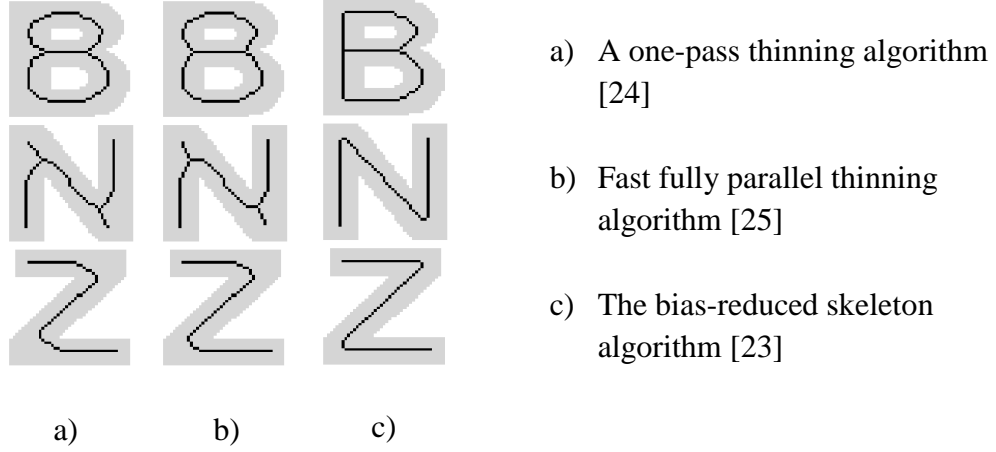


Figure 1.3 Skeletons of the letters B, N and Z created by different skeletonization algorithms [6]

### 3D reconstruction

Because the Timpix3 detector is capable of capturing the particle energy, we are able to reconstruct its trajectory in 3D. We can imagine each pixel of Timepix3 detector as a cylinder or a rectangular block of non-zero height. When a particle traverses the detector, it emits charge carriers (holes) along the way, which drift toward the electrodes of the detector and cause the pixels to capture some non-zero energy. However, Timepix3 stores data about the time of arrival which can be used to estimate the relative z-coordinate of the particle in the space (relative to the first registered hit – the pixel with minimal  $ToA$  has  $z$  set to 0). The  $z$  coordinate is a function of the relative time of arrival  $t_r$ , where  $t_r$  is the difference between the arrival time of the particular pixel and the minimum time of arrival in the entire cluster [7]. It holds:

$$z(t_r) = \frac{d}{U_d} (U_d + U_b) \left[ 1 - \exp \left( 2 * \frac{2U_d\mu_h}{d^2} t_r \right) \right] \quad (1.1)$$

Parameters used are the following:

- $U_b$  – depletion voltage that is the minimum voltage at which the bulk of the sensor is fully depleted,
- $U_b$  – bias voltage, which is the amount of voltage that a detector needs in order to function [8],

- $\mu_h$  – electrical mobility of a hole, and
- $d$  – thickness of the sensor.

All parameters are specified at the beginning of the measurement and remain constant for the whole duration of the measurement.

## 1.4 Classification

After we are able to visualize clusters and analyze their properties, we can start focusing on subsequent classification. Firstly we briefly analyze the work done in terms of cluster classification. Then, we discuss the problems and the choice of the classifier model. Another topic we examine is the generation of the training data.

Classification of a cluster is a task where we are given a cluster and a set of possible classes. Based on the cluster, we should predict to which class the cluster belongs. Ideally, we also want to estimate how sure we are about our prediction and possibly return the result "unclassified" if we are not confident about the prediction.

So far, there have been a few attempts to classify clusters – one example being the work *Detecting elementary particles with Timepix3 detector* [9]. In this work, the clusters were divided into classes based primarily on their shape. The categories used in the thesis were the following:

- **Dots** consist of up to four pixels.
- **Heavy blobs** come with a round shape with the most energy located in the center while having a significant halo effect.
- **Long gammas** tend to have a straight shape and contain only a few pixels (less than 20 pixels).
- **Straight tracks** represent tracks of particles with minimum ionization. These tracks also have a linear shape, but they are longer than long gammas (more than a hundred pixels).
- **Curly tracks** are the tracks with the shape of a nonlinear curve.
- **Heavy tracks** correspond to the heavy ions that traverse the detector at a non-perpendicular angle. Similar to heavy blobs, heavy tracks leave many halo effect pixels. These tracks also have a very high total energy.

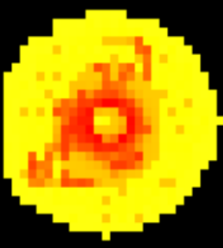



• Dot		• Heavy Blob	
• Long gamma		• Straight track	
• Heavy track		• Curly track	

Table 1.2 Categories of clusters based on their shape

Even though this classification provides valuable information about the shape of a cluster, mapping these categories to the real particle examples is still non-trivial. For instance, one particle can have a dot shape when it traverses the detector perpendicularly. In contrast, if the same particle enters the detector's field in a direction parallel with the orientation of sensors, it could leave a curly track, as shown in Figure 1.4.

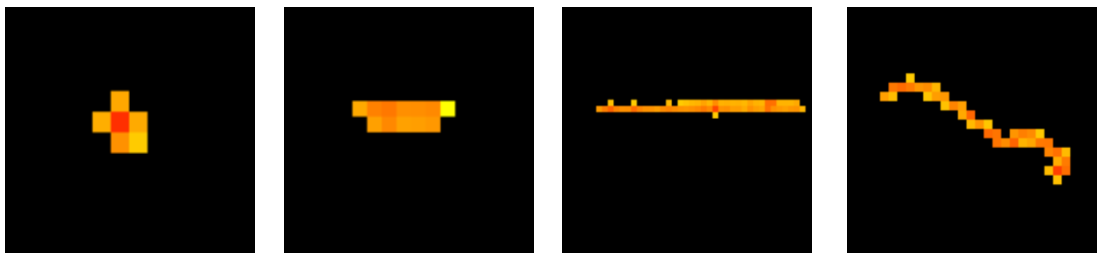
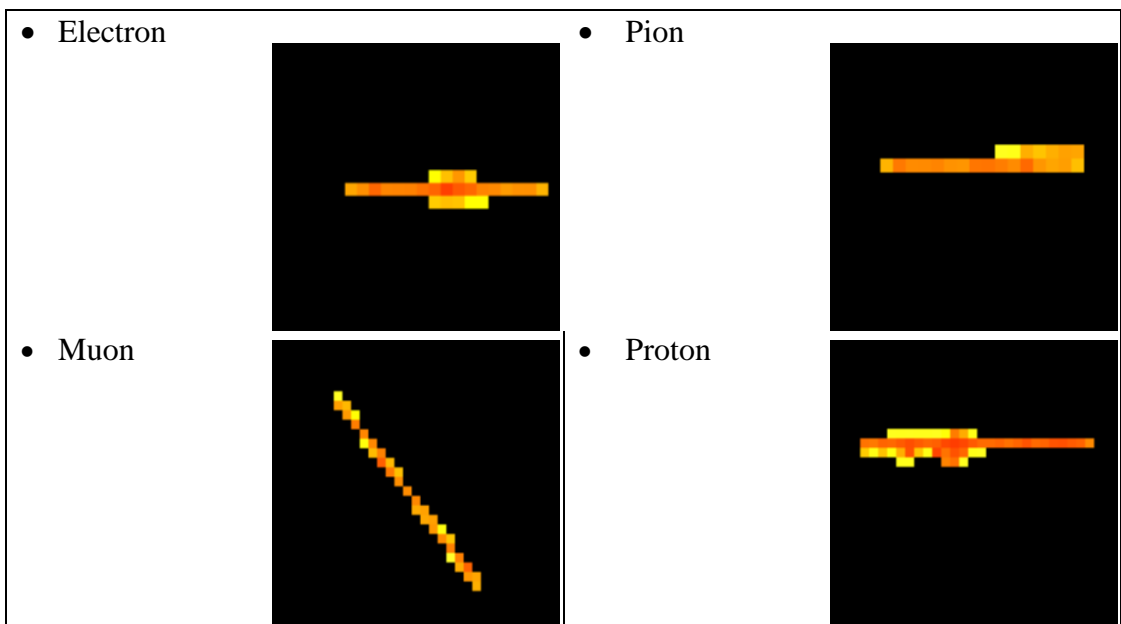


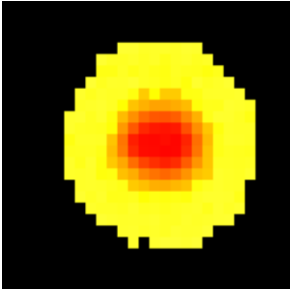
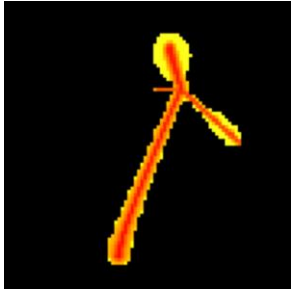

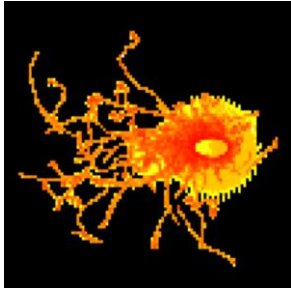

Figure 1.4 Possible 2D visualizations of an electron

The categories we decided to use for classification were based on the training data we were provided by Mr. Declan Garvey from the Institute of Experimental and Applied Physics of the Czech technical university. These data were obtained from the simulations, which could potentially affect the classifier quality on the real (non-simulated) dataset. The classes of the particles in the data were the following:

- **Electron, Pion, and Muon** are the particles that have a relatively small total energy. Often they leave a track with a linear shape and have an energy of more than 10keV.
- **Low energy electron** usually has a shape of a dot. These electrons tend to have a total energy of less than 10keV.
- **Proton** cluster tends to lead a linear track. Compared to electrons, it usually has significantly higher total energy and pixel count.
- **Helium** is a cluster that often comes with a non-trivial halo. Its energy is mainly concentrated in the center of the cluster.
- **Fragment** is a cluster that can contain multiple straight and curly trajectories with high energy.
- **Iron** cluster is usually round. It differs from helium by its higher energy and curly trajectories with high energy coming from its center.
- **Lead** clusters are the largest in our dataset. They often have tens of branches, the largest pixel count, and energy.





• Helium		• Fragment	
• Iron		• Lead	
• Low energy electron			

**Table 1.3 Types of the clusters for classification with their examples**

Because it seemed to be very difficult to set the criteria for each class manually, we decided that a machine-learning-based approach could be a good choice. We chose neural networks as these became very popular when it comes to solving complex problems, and in many tasks, they achieve state-of-art performance. When fed with the data, the neural network model can learn from the data features until it reaches its maximum accuracy. There are many kinds of neural networks, but we narrowed the choice down to the two primary candidates – the convolutional neural network (CNN) and the multi-layered perceptron (MLP). The first candidate – the CNN – is widely used for the 2D image analysis, but it has a couple of drawbacks for our task. For instance, the cluster would have to be represented as a 2D image containing both the pixels with non-zero energy and also the ones carrying no energy. Hence, using CNN could be a little less efficient because CNN processes all of the given pixels (even though the zero energy pixels provide no information about the cluster). Considering the fact that we are able to calculate features and work with

them instead of the whole set of pixels leads us to the conclusion that we could use the feature-based MLP model.

### *Training Data Generation*

Even though we had the data to train our model, the data were separated into files by particle type and angle of crossing the detector. This format can be great for viewing and browsing, but it is preferable to have one data source for training purposes. For that, it is necessary to create an application that processes multiple files and generates a single training collection suitable for training a neural network. When creating this application, we need to keep in mind that our dataset is likely to have an uneven distribution of classes. Unbalanced training data cause many problems in training classifiers. E.g., let us consider training data with samples from two classes  $A$  and  $B$  where we have 99% of training samples from class  $A$  and only 1% from class  $B$ . Then a trivial classifier classifying all samples as being from class  $A$  has high accuracy (99%), but it can not detect any particle from class  $B$ . So we need to add an option to prepare a balanced training set with an even distribution of classes.

## 2 Goals of the thesis

Timepix3 detector can produce a vast amount of data on elementary particles. We aim at supporting physicists working with such data. Based on the observations from Chapter 1, we can formulate the main goals of the thesis:

- *Cluster visualization* – The commonly used MM file format (Table 1.1) provides plenty of machine-readable information about the clusters. However, it can be difficult for a physicist to extract an intuitive overview of the cluster dataset from this format. That is why we decided to set the first goal to **design an interactive cluster visualization tool, which would display various properties of a single cluster (like a 2D and 3D image of a cluster) while also displaying a brief overview of the whole dataset.**
- *Cluster filtering* – Because we expect the input data to contain regular and uninteresting clusters mostly, we might need to select non-trivial clusters with some prescribed properties, like a minimal number of pixels. In order to do that, **we must create an application for filtering based on the attributes of a cluster. This application should be efficient as we expect it to process a large amount of input data.**
- *Classification* – Because of the lack of tools for cluster classification, **we would like to create an interface supporting the classification of clusters by using some machine-learning methods.** Machine learning is the ability to learn to solve tasks from data. Here we propose to support the training of classifiers able to classify clusters based on classified samples. Such an approach has a big advantage as it can be adapted to different types of particles and different types of data. E.g., if the particles are detected under new conditions, we could train a new classifier adapted for the new type of data. In order to develop such a classifier tool, we set two additional subgoals:
  - *Data preparation* – Before we start with the classification, **we must create an application for preparing the training data for the subsequent classification.**
  - *Classifier training* – Because physicists need to analyze different kinds of cluster datasets, we would like not only to

create a single tool aimed at the classification of the specific clusters. Instead, **we want to create a parameterized interface for classifier training, which would enable the users to train and customize the classifier models based on their needs and the specific dataset.**

- *Evaluation* – To show that our developed tools are applicable, **we will design experiments to evaluate the quality of classifiers trained on sample data.**

## 3 Cluster processor applications

In this chapter, we will describe all of the applications that we created in order to address the goals specified in Chapter 2. First, in Section 3.1, we go over the brief specification of the whole solution. Then, in Section 3.2, we introduce several types of neighborhoods later in Section 3.3, where we describe calculations of cluster features. Next, in Section 3.4–3.7, we proceed with a more detailed discussion about each specific application, its functionality, and its essential features.

### 3.1 Specification

Based on the observations from Chapter 1, we decided to develop a solution consisting of multiple applications – tools for viewing, analyzing, and classifying the clusters. We split the solution into multiple applications because this way allows for better extendability. When some developer comes with a better idea for the classification, he can still use the rest of our solution and adjust only some part of the application. Another advantage of this approach is that each user can decide what functionality is required and download only the specific projects that cover the user's requirement. These tools are primarily made for physicists and could be divided into interactive and non-interactive. An interactive tool can be used mainly for the visualization of one cluster at a time. Non-interactive tools are used for unattended processing of large datasets:

- ***Interactive tools:***
  - ***ClusterViewer*** is an application for cluster visualization. The user can select a dataset containing clusters in MM format and randomly select and inspect one cluster at a time. ClusterViewer can display both 2D and 3D visualizations of the cluster and its skeleton. Furthermore, the viewer can load a trained classifier model and apply it to the selected cluster. Besides showing properties of the selected cluster, the application presents histograms of cluster properties from the whole dataset.
- ***Non-interactive tools:***

- ***ClusterFilterer*** is a tool for cluster filtering based on its features. The user selects the desired interval for values of the attributes in a dialog form, and then the whole collection of clusters is processed at once.
- ***DescriptionGenerator*** is a program that processes multiple input files into one training dataset based on the options selected in a user interface.
- ***ClusterClassifier*** is a console application that applies a trained classifier on a dataset of clusters in JSON format. The output of the application is a classification of the whole dataset and splitting the data into multiple files based on the predicted class attribute.
- ***ClusterClassifierUI*** is a user interface where it is possible to train a classifier when provided classifier configuration parameters and the training data. Apart from training, the user can also merge trained classifiers into multi-level classifiers.
- ***ClusterTrainer*** is a console application capable of training a classifier after configuration parameters and training data are provided.

## 3.2 A little about neighboring

Neighboring of the pixels is an essential term when it comes to 2D image analysis. In general, two pixels are called neighbors if there is a relatively small distance between them. To be more specific, there are used two kinds of neighbors:

- ***8-neighbors of a pixel  $p$***

$$\{n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$$

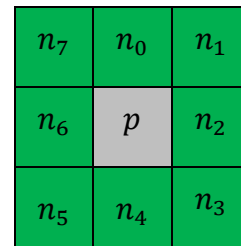


Figure 3.1 8-neighbors of a pixel  $p$  (green color)

- *4-neighbors of a pixel  $p$*

$$\{n_0, n_2, n_4, n_6\}$$

$n_7$	$n_0$	$n_1$
$n_6$	$p$	$n_2$
$n_5$	$n_4$	$n_3$

Figure 3.2 4-neighbors of a pixel  $p$  (green color)

And for future use, we will also introduce another kind of neighbors:

- *Y-neighbors of a pixel  $p$*

This kind of neighbor has two main variants:

Variant A  $\{n_1, n_4, n_7\}$

Variant B  $\{n_0, n_2, n_5\}$

$n_7$	$n_0$	$n_1$
$n_6$	$p$	$n_2$
$n_5$	$n_4$	$n_3$

Figure 3.3 Y-neighbors - variant A (green color)

$n_7$	$n_0$	$n_1$
$n_6$	$p$	$n_2$
$n_5$	$n_4$	$n_3$

Figure 3.4 Y-neighbors - variant B (green color)

For each variant, we also consider their symmetric alternatives to belong to the same variant.

### 3.3 Calculating the features

A cluster can be defined as a set of pixels, each containing information about its  $x$  and  $y$  coordinates,  $ToA$ , and energy. Apart from these explicit features, we also analyze the implicit features of the cluster, like pixel count and the cluster's total energy. This is the critical point of all our following work and also the main topic of this subchapter. We briefly discuss the features and the information they indicate

about the cluster. The properties we can calculate range from straightforward to more sophisticated. The most important features are the following:

- **Total, average and maximum energy, the standard deviation of energy,** and **low energy pixels count** provide information about the energy distribution of pixels in a cluster. The total energy of a cluster is defined as the sum of the energies of each pixel in the cluster. A pixel is considered to have low energy if its energy is less than 10keV.
- **Pixel count** attribute reflects the size of a cluster.
- **Width (diameter)** and **convexity** attributes are both based on the convex hull of a cluster. Let  $V_1, V_2 \dots V_n$  denote the vertices of the convex hull of a cluster. Then the convexity is defined as follows:

$$Convexity = \frac{Pixel\ count}{Area\ of\ the\ polygon\ V_1V_2\dots V_n}$$

The convexity of a cluster provides us the information about its shape because the more complicated clusters usually tend to have a concave shape. In contrast, the simple ones are often more convex. Width, also commonly known as the diameter, is defined as  $\max_{\substack{i \in 1..n \\ j \in 1..n}} d(V_i, V_j)$  where  $d$  represents the distance between given vertices. For the function  $d$ , we chose to use the Euclidian distance in a plane:  $d(V_i, V_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$  where  $V_i = (x_i, y_i)$  is a pixel with the coordinates  $x_i$  and  $y_i$ .

- **The standard deviation of the ToA** captures the information about the timespan of a cluster. Another option would be to use the difference of the  $ToA_{max} - ToA_{min}$ . This, however, has a disadvantage because, in a cluster, there are often a few pixels that show extreme values of ToA. The standard deviation is considered to be "more resistant" to these outlier pixels and provides a reasonable estimate of the timespan of the cluster.
- **Vertex count, Crosspoint count,** and **Branch Count** reflect the possible number of particles in a cluster and the shapes of the trajectories in the cluster. A pixel is considered to be a *vertex* if it only has one neighboring pixel in the cluster.



Crosspoints are the pixels where the trajectories of different particles meet. These pixels we find as the ones with non-zero energy and with three or more 4-neighbors or there is any Y-neighbourhood (among four rotations of both Variant A and Variant B) in which the pixel has three neighbors. Crosspoints of a set  $C$  are denoted by  $Cross(C)$ . Figure 3.5 shows a cluster with four crosspoints.

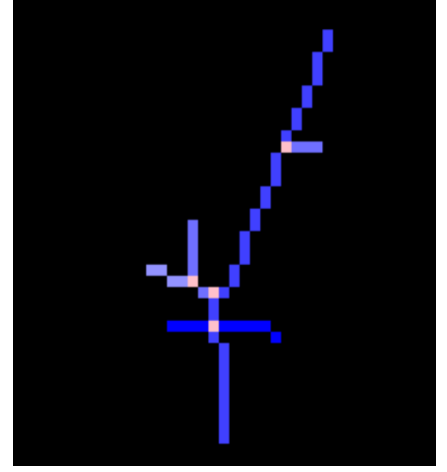


Figure 3.5 A cluster with four crosspoints (pink color)

Because we do the branch analysis iteratively from the center, we define the term *branch found at the  $k$ -th iteration* instead of a simple *branch*. A branch found at the  $k$ -th iteration is a set of pixels defined as follows: Let  $P$  denote a set of pixels in a cluster  $C$ . Let  $G = (P, E)$  denote a graph with the set of vertices  $P$  and the set of edges  $E$  such that there exists an edge between pixels  $p_1$  and  $p_2$  in  $E$ , if  $p_2$  is an 8-neighbor of  $p_1$ . Then we say  $P$  is a branch found at the  $k$ -th iteration if and only if its corresponding graph  $G$  is a path,  $P$  has the maximal possible size, and at least one of the following conditions holds:

1.  $k = 0$  (base step of the definition), or
2. none of the pixels in  $P$  are part of any of the previous  $k - 1$  branches in  $C$  (inductive step of the definition)

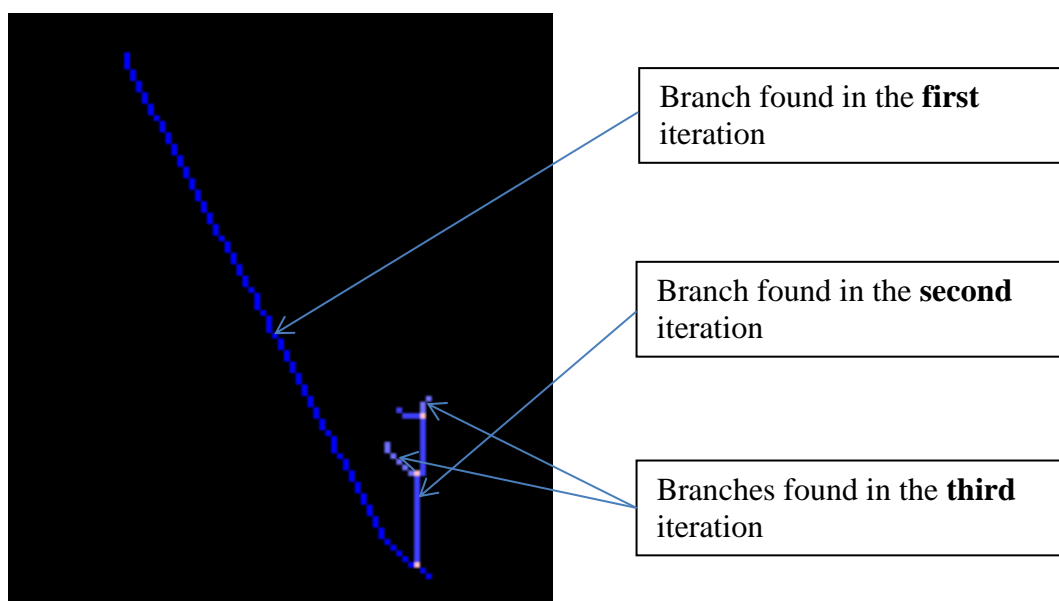


Figure 3.6 Branches of a cluster found at different iterations have different color

### 3.4 Cluster Viewer

In this subchapter, we will discuss all the features of the application for cluster visualization – **ClusterViewer**. This application contains tools to display and analyze properties of a cluster. The features range from a simple 2D image to a 3D image and the analysis of the particle class.

#### *Input*

Input for the **ClusterViewer** application is a single **ini** file in the MM file format (described in Table 1.1). The **ini** file is selected via the user interface using either a dialog window or by typing the path to the file. Another option is to load a file in JSON file format, which is an array of objects (clusters). This file format is an output of the **DescriptionGenerator** application and can also be an output of the **ClassifierForClusters** program. Each of the objects in the JSON array must contain three specific keys in order to be successfully loaded. These are:

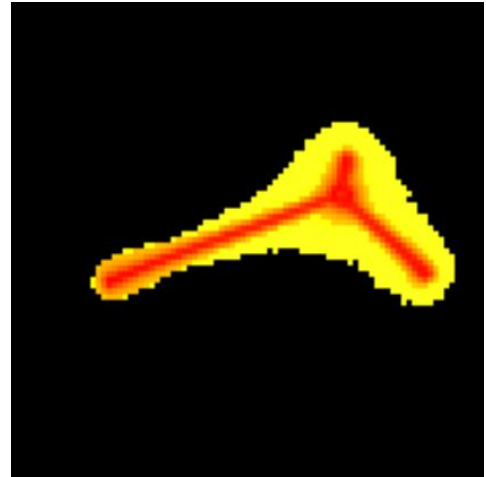
- **ClFile** – a relative path to the cluster **cl** file, starting from the directory where the output file is located (it is recommended to not move the created output file in order to track the original **cl** file successfully),

- **PxFile** – a relative path to the cluster **px** file (similar to **ClFile**), and
- **ClIndex** – index of the given cluster in the **cl** file.

Once the file is successfully loaded, we can proceed to the other tools.

## ***2D view***

To get a better overview of the cluster dataset, we decided to create a 2D image of the cluster's energy. The primary purpose of the image is to visualize the cluster as a bitmap, assigning colors based on the energy of the pixels. The viewer contains a 2D image that is represented as a 256×256 bitmap. In the cases when the same pixel is hit multiple times in a cluster, we decided to display the hit with the highest deposited energy. Because the low-energy pixels prevail in most clusters, we chose



**Figure 3.7 Cluster visualization indicating the halo pixels (yellow)**

to map energies to color space logarithmically, which seemed to distinguish the energies without the need for a wide range of colors. For a more complicated cluster, this could also distinguish between halo pixels and the "real" particle trajectory. Users can navigate through the currently loaded collection of clusters by clicking **Previous** and **Next** buttons or typing in the index of a cluster and clicking the **Find** button. Using the mouse wheel is also possible to zoom in the 2D view of the cluster based on the position of the mouse pointer.

## ***Collection Histogram***

Another feature of the application is computing and presenting the **Collection Histogram**. That is the histogram of the currently loaded collection of clusters, representing the distribution of clusters with respect to **PixelCount**.

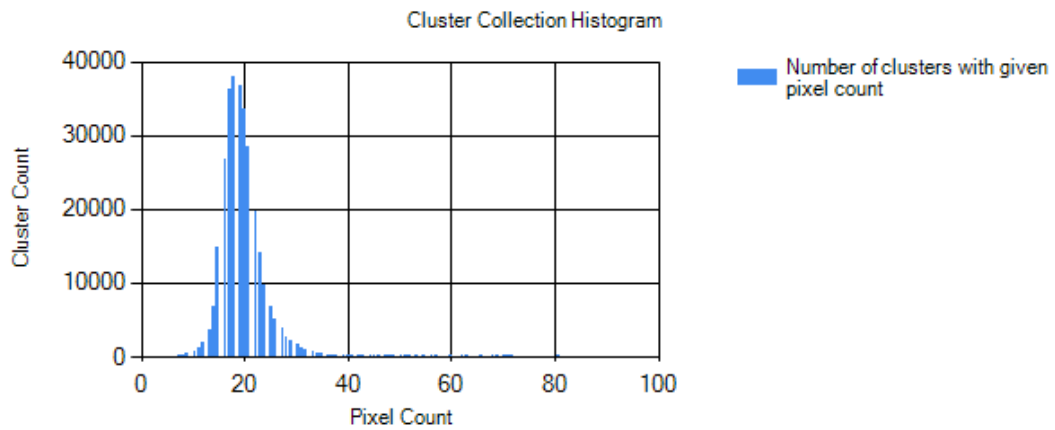


Figure 3.8 Histogram of the clusters with respect to their pixel count

The default option will display the histogram based on the cluster's pixel count, but the program can be easily extended to display the histogram of any feature computable for each cluster.

### ***Pixel Histogram***

Pixel Histogram works similarly to the Collection Histogram, except for the fact that it depicts the histogram of the pixels in the currently loaded cluster. The default displayed property of the pixel is its energy. This histogram could be helpful when classifying given cluster because similar classes tend to have a similar energy distribution.

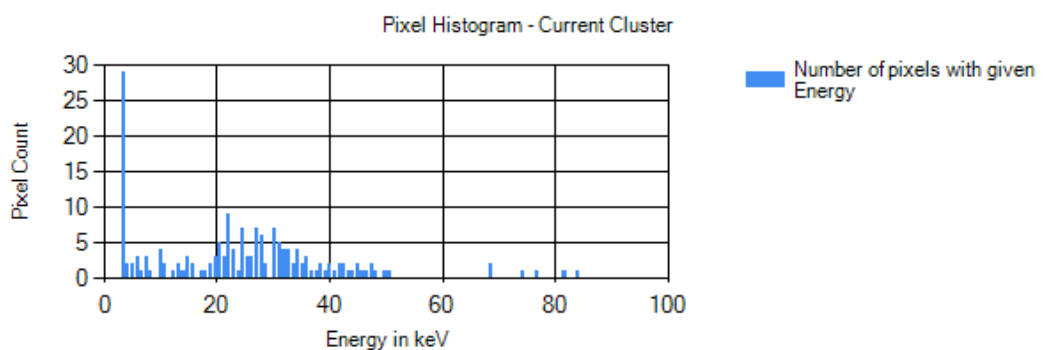


Figure 3.9 Histogram of pixels in a single cluster based on the energy of each pixel

### 3D Visualization

Based on the ToA feature of the cluster, we calculated the  $z$  coordinate according to the formula (1.1) for the dependency of the ToA from the  $z$  coordinate of the pixel. This way, we transformed two-dimensional points into three-dimensional, which we then showed visually as a 3D scatter plot. For a better viewing experience, we added an option to rotate the image around the  $x$  and  $y$  axes.



Figure 3.10 2D visualization of the cluster with linear trajectory

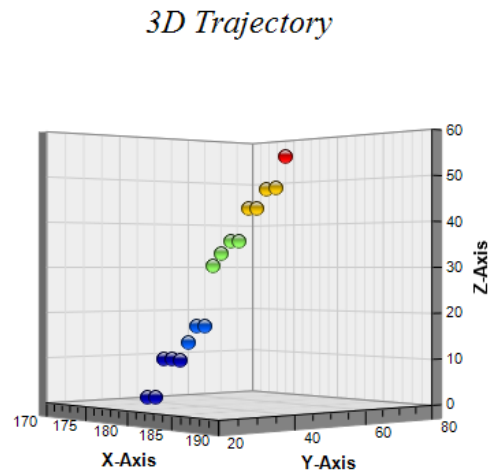


Figure 3.11 3D visualization of the cluster with linear trajectory from Figure 3.10

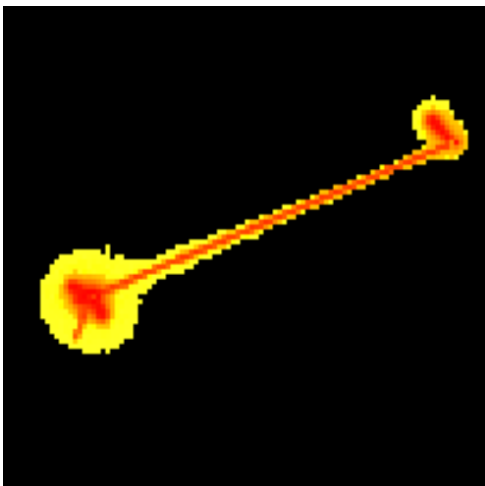


Figure 3.12 2D visualization of the cluster with multiple branches

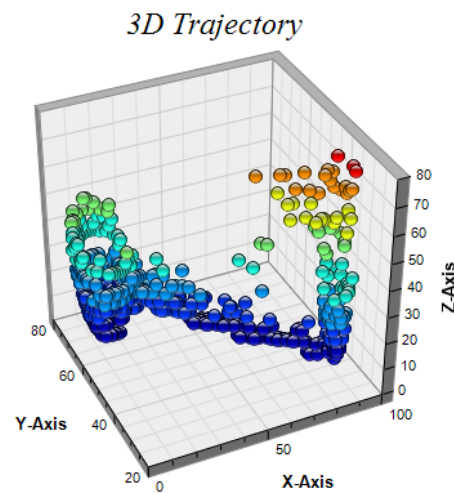


Figure 3.13 3D visualization of the cluster with multiple branches from Figure 3.12

## Skeletonizer

Skeletonization, as introduced in Section 1.3, simplifies the cluster while maintaining its geometric properties. Therefore, we apply it as the first step in extracting the possible particle trajectories in a cluster. To find a skeleton of a cluster, we decided to modify Zhan-Shuen's algorithm for thinning binary digital patterns [10] in the following way:

---

### Algorithm 3.1 Skeletonization of a cluster

---

	<b>Function</b> Skelet( $S, t$ )	# $S$ is a set of pixels
1.	$D_1, D_2 \leftarrow \emptyset$	
2.	<b>do</b>	
3.	<b>For each</b> pixel $p$ in $S$	#first sub-iteration
4.	<b>If</b> $conditionA(S, p)$ <b>and</b> $p.energy < t$	
5.	$D_1 \leftarrow D_1 \cup \{p\}$	
6.	$S \leftarrow removeAndSplitEnergy(S, D_1)$	
7.	<b>For each</b> pixel $p$ in $S$	#second sub-iteration
8.	<b>If</b> $conditionB(S, p)$ <b>and</b> $p.energy < t$	
9.	$D_2 \leftarrow D_2 \cup \{p\}$	
10.	$S \leftarrow removeAndSplitEnergy(S, D_2)$	
11.	<b>while</b> $D_1 \cup D_2 \neq \emptyset$	
12.	<b>return</b> $S$	

Variables and functions used in skeletonization	
$S$	The set of pixels remaining in the skeleton
$D_1, D_2$	Sets of pixels to delete after each sub-iteration
$N(p, S)$	The set of 8-neighbors of $p$ in $S$
$n_i(p, S)$	true if the $i$ -th neighbor of $p$ in $S$ has non-zero energy; for numbering the neighbors see Figure 3.14
$p.energy$	The energy of the pixel $p$

$t$	Chosen threshold energy
$removeAndSplitEnergy(S, D)$	Remove the pixels in $D$ from $S$ one by one and split their energy evenly across their neighbors in $S$
$K_{01}(p, C)$	Each neighbor $n_i$ of a pixel $p$ is assigned a value 0, if $n_i.energy = 0$ , and 1 otherwise. $K_{01}(p, C)$ is then the number of 01 occurrences in the sequence of neighbors of $p$ in $C$ ( $n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8$ ) as displayed in Figure 3.14. Each neighbor $n_i$ is assigned a value 0 if $n_i.energy = 0$ and 1 otherwise.
$conditionA(S, p)$	$ N(p, S)  > 1$ and $ N(p, S)  < 7$ and not( $n_0(p, S)$ and $n_2(p, S)$ and $n_4(p, S)$ ) and not ( $n_2(p, S)$ and $n_4(p, S)$ and $n_6(p, S)$ ) and $K_{01}(p, S) = 1$
$conditionB(S, p)$	$ N(p, S)  > 1$ and $ N(p, S)  < 7$ and not( $n_0(p, S)$ and $n_2(p, S)$ and $n_6(p, S)$ ) and not ( $n_0(p, S)$ and $n_4(p, S)$ and $n_6(p, S)$ ) and $K_{01}(p, S) = 1$

Table 3.1 Description of functions and variables used in Algorithm 3.1.

In the first sub-iteration of Algorithm 3.1, we remove pixels with energy lower than the threshold which are either east or south boundary point, or a point in the northwest corner of the cluster (represented by *conditionA*). On the contrary, in the second sub-iteration, we remove points on the north and west boundaries and the points on the cluster's southeast corner (represented by *conditionB*). We repeat the process while at least a single pixel was deleted in any of the iterations. The whole skeletonization process consists of using Algorithm 3.1 twice. The first time with value  $t$  set to 10 keV and  $S$  set to all pixels in cluster and then we

$n_7$	$n_0$	$n_1$
$n_6$	$p$	$n_2$
$n_5$	$n_4$	$n_3$

Figure 3.14 Example of pixel  $p$  with  $K_{01} = 3$  (the number of times the color changes from white to green in the ordered sequence of its neighbors). White color means 0, and green means 1.

repeat the process with  $t$  set to infinity and  $S$  set to the return value from the first iteration of the skeletonization algorithm. As a result, the first iteration aims to remove the halo effect of the low-energy pixels while still preserving the shape. The halo effect is created when the particles of a cluster have high energy, so apart from the real position of the particle also neighboring pixels capture non-zero energy. The second iteration of the algorithm with  $t = \infty$  behaves the same as the original version of Zhan-Shuen's algorithm [10]. A different approach would be to filter the low-energy pixels right away, but that does not guarantee that the cluster remains connected, which is a problem for future branch analysis.

The application can also apply skeletonization to the currently viewed cluster. In contrast to the original thinning algorithm, our modified algorithm preserves more information about the shapes of high-energy trajectories. The differences in the output of the original skeletonization algorithm and our modification can be seen in Figure 3.15, Figure 3.17, and Figure 3.16.

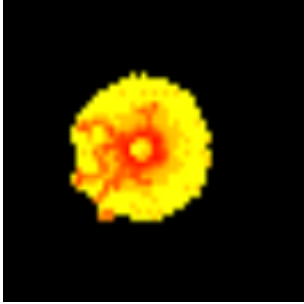


Figure 3.15 Original cluster

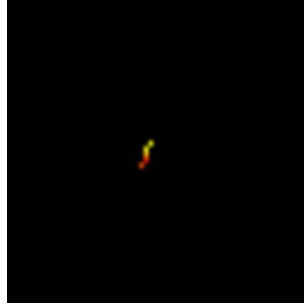


Figure 3.17 Skeletonized cluster (original Zhan Shuen's algorithm)

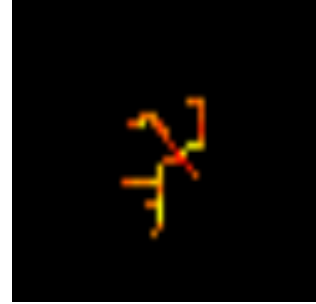


Figure 3.16 Skeletonized cluster (modified Zhan Shuen's algorithm)

### *Center Finder*

When analyzing the shape of a cluster, it could be useful to know where the center of the cluster event is. The center is usually the point with high energy located at the intersection of the visible trajectories (if there is any). If we manage to find this point correctly, we could then start analyzing each trajectory starting from the core pixel of a cluster event. However, this task proved to be quite challenging, especially for the complicated clusters, because there can be multiple points with a high energy level lying at the intersection of the trajectories. For this task, we proposed an algorithm based on the energy of the pixel and its surrounding pixels: Let  $C$  denote an arbitrary set of pixels and let  $S_c(p)$  be the set of pixels surrounding the pixel



$p \in C$  excluding the pixel  $p$  (possibly  $S_C(p) = \{n \in C \mid 0 < d(n, p) < \varepsilon\}$  where  $d$  can be the Euclidian distance and  $\varepsilon = 3$ ). Then, we define the weighted surrounding energy of a pixel  $p \in C$  with the weight  $w_C$  as  $E_{S_C}(p, w_C) = w_C \sum_{n \in S_C} E(n)$  with  $E(n)$  being the energy of a neighboring pixel  $n$ . For each pixel, the surrounding energy is used to calculate the center cost function.

Let  $C$  represent the set of pixels of a cluster and let  $T$  denote its skeletonized version. The center cost function is defined as follows:

$$f(p) = E_{S_C}(p, w_C) + E_{S_T}(p, w_T) + w_p E(p)$$

In our algorithm, we set  $w_C = 1, w_T = 1.5, w_p = 1.5$ . Then the center pixel of a set of pixels  $C$  is computed simply as:

$$p_{center} = \begin{cases} \operatorname{argmax}_{p \in \operatorname{Cross}(C)} (f(p)) & \text{if } \operatorname{Cross}(C) \neq \emptyset \\ \operatorname{argmax}_{p \in C} (f(p)) & \text{otherwise} \end{cases}$$

### **Branch Analyzer**

Branch analysis is the process of finding the branches in a cluster. We say the branch  $s$  is a subbranch of the branch  $b$ , if it starts in one of the crosspoints of branch  $b$ . To analyze the branches, we proposed the following algorithm:

---

#### Algorithm 3.2 Branch analysis

---

```

function BranchAnalysis( $C$ )      #  $C$  is the set of pixels in a cluster
1.   $U \leftarrow \operatorname{Skelet}(C), s_C \leftarrow \operatorname{Center}(S), Bs \leftarrow \emptyset$ 
2.  do
3.     $B \leftarrow \operatorname{GetBranch}(U, s_C, \operatorname{Cross}(U), \operatorname{maxDepth})$ 
4.     $Bs \leftarrow Bs \cup \{B\}$ 
5.     $U \leftarrow (U - B.\operatorname{TotalPoints}) \cup \{s_C\}$ 
6.  while ( $|B.\operatorname{TotalPoints}| > \operatorname{trivialBranchSize}$ )
7.     $\operatorname{AddMissedCrosspoints}()$ 
8.     $Bs \leftarrow \operatorname{SortByPointsCountDesc}(Bs)$ 
9.    foreach pair  $(B_1, B_2)$  in  $Bs$ 
10.     if ( $\operatorname{AngleOfSplit}(B_1, B_2) < \operatorname{epsilonAngle}$ )
11.        $\operatorname{MergeBranches}(B_1, B_2)$ 
12.     break()
13. return  $Bs$ 

```

---

**Algorithm 3.3 Get Branch function**

---

```
function GetBranch( $U, s_C, U_{Cross}, d$ )  
1.  $B \leftarrow \text{FindLongestPathBFS}(U, s_C)$   
2.  $L_{Cross} \leftarrow (B \cap U_{Cross}) - \{s_C\}$   
3.  $P_{closeToStart} = N(s_C) - L_{Cross}$   
4.  $P_{Forbid} = ((B - L_{Cross}) \cup P_{closeToStart} \cup \{s_C\})$   
5.  $U_{new} = U - P_{Forbid}$   
6. If ( $d > 0$ )  
6.   foreach  $p_{Cross}$  in  $L_{Cross}$   
7.      $B.\text{addSubBranch}(\text{GetBranch}(U_{new}, p_{Cross}, U_{Cross}, d - 1))$   
8. return  $B$ 
```

<i>Variables and functions used in Algorithm 3.2 and Algorithm 3.3</i>	
$U$	The set of points we can use for the branch analysis
$s_c$	The start point of the current branch
$U_{cross}$	All crosspoints
$d$	Maximal depth of the algorithm search
$Skelet(C)$	The result of the skeletonization algorithm (Algorithm 3.1)
$Center(C)$	The center of a cluster $C$
$RemoveDuplicates()$	Removes duplicate points (comparison by $x$ and $y$ coordinates)
$B.TotalPoints$	Union of all points of all subbranches of $B$ with itself
$MergeBranches(B_1, B_2)$	Merge two longest branches into a single branch
$FindLongestPathBFS()$	It numbers the pixels using BFS from the start of the branch and returns the path from the pixel with the highest number to the start of the branch
$maxDepth$	Maximal depth of the algorithm, the number of times $GetBranch$ can be called recursively, effectively speeds up the calculation on large clusters
$trivialBranchSize$	The size of a branch which is too small to provide us any information (less than 2 pixels)
$SortByPointsCountDesc()$	Sorts branches by their size in descending order
$AngleOfSplit(B_1, B_2)$	The angle between the branches (up to the first 10 points of the branch are approximated using linear regression, and then the angle is calculated between the lines from the regression)

Figure 3.18 Variables and functions used in branch analysis algorithms

To find branches in Algorithm 3.2, we first compute the skeleton and the center of a cluster. Then, we find a path to the most distant point from the center in the skeleton using a breadth-first search. While we are searching for the path to the most distant point, we also find the crosspoints on that path, which we use for the recursive analysis of the sub-branches. Before calling the recursive function for finding the branches, we need to temporarily remove the points in the skeleton, which are already part of some branch (to make sure the same points do not belong to more branches). The recursion is stopped if there are no crosspoints in the current branch or the depth of the recursion is higher than the maximal allowed depth. We also used this algorithm (Algorithm 3.2) to analyze branches in the `ClusterViewer` application. We analyze the given cluster and search for possible trajectories of various particles captured in the cluster. These trajectories – branches – are then distinguished by their colors. A branch is considered to be the main branch if it starts in the center of a cluster. Each branch can also have its subbranches, which are denoted by the lighter version of their parent's branch color. For an example of the branch analysis on a non-trivial cluster, see Figure 3.20 and Figure 3.19.

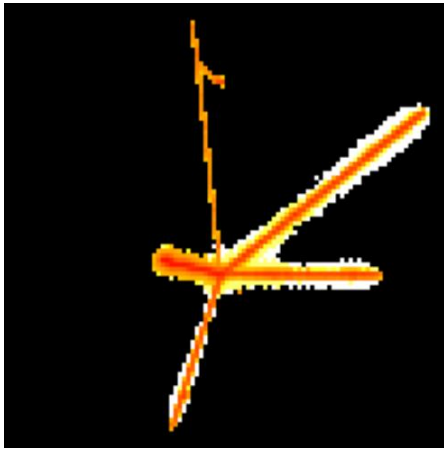


Figure 3.20 Original cluster before branch analysis

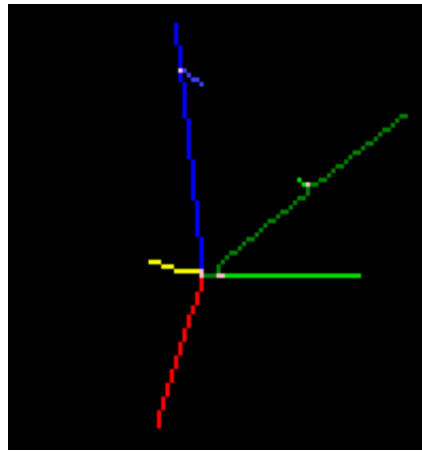


Figure 3.19 Cluster after branch analysis, each branch with its distinct color (pink points are the cross points where the branches split)

### 3.5 Cluster Filter

After we created `ClusterViewer` application, we noticed that a vast majority of the clusters only had a few pixels. In order to find some non-trivial clusters, we created `ClusterFilter` application. It consists of a simple user interface, where a user selects the `ini` file and the upper and lower bound for the attributes of a cluster, see Table 3.2. After that, only the clusters which fit into the interval (lower bound,

upper bound) are selected and written into a new `c1` file. During the process, `px` file is not copied. Instead, the former `px` file is used, so this newly created `c1` file only provides "a new view" into the original `px` file.

In order to speed up the computation, we implemented the lazy evaluation, which means that after some filtering

condition for the cluster fails, the rest is not evaluated. Attributes are calculated in order from the easiest to calculate to the most expensive. This implies that setting some easy-to-compute attribute to a narrow interval could greatly decrease the computation time, so we recommend doing so any time it is possible.

Select filters		
<b>TotalEnergy</b>	From: <input type="text"/>	To: <input type="text"/>
<b>MaxEnergy</b>	From: <input type="text"/>	To: <input type="text"/>
<b>PixelCount</b>	From: <input type="text"/>	To: <input type="text"/>
<b>VertexCount</b>	From: <input type="text"/>	To: <input type="text"/>
<b>Convexity</b>	From: <input type="text"/>	To: <input type="text"/>
<input type="checkbox"/> Use skeletonized convexity		
<b>Width</b>	From: <input type="text"/>	To: <input type="text"/>
<b>BranchCount</b>	From: <input type="text"/>	To: <input type="text"/>

Table 3.2 User interface of the filterer - setting the lower and upper bound for filters

## 3.6 Description Generator

The `DescriptionGenerator` is the application for preparing data (the features of a cluster) for the consequent training of the machine-learning-based classifier model. The application requires the data to be in the MM (clustered) format Table 1.1. We also made the algorithm extendable to more features without requiring a lot of programming.

### *Input selection*

First, a user selects and loads one or more `ini` files via the dialog window. All of the currently loaded files are displayed in the UI, as shown in Figure 3.21.

Each such loaded `ini` file is called a partition. A partition can be assigned a class name, which is an editable field in the left column of the displayed table. These partitions are later grouped based on the class names to form the classes. The class

Selected Input	
Class Name (optional)	Selected File Path
<none>	D:\source\repos\Celko 2020 Example data\data\particles\frag\16_nuc.ini
<none>	D:\source\repos\Celko 2020 Example data\data\particles\pions\deg45\3_1.ini
<none>	D:\source\repos\Celko 2020 Example data\data\particles\pions\deg45\g3_1.ini
<none>	D:\source\repos\Celko 2020 Example data\data\particles\pions\deg45\k4_0.ini

Figure 3.21 Loaded ini files into the DescriptionGenerator

names are compared as strings, so one needs to keep in mind the names are case-sensitive. Apart from choosing input files, a user must also select the name for the output file.

### Feature selection

After the files are selected, a user can choose which attributes will be calculated by selecting the corresponding item in the displayed list of features. The algorithm is designed only to calculate the features that are selected. Even though more features might provide more information about the cluster, adding features could also increase the algorithm's time complexity and, therefore, the duration of the whole calculation. We calculate features for each cluster from the input and store them into a single file in the JSON file format. The feature calculation complexities can be split into four categories:

1. Expensive:
  - BranchCount,
  - Branches.
2. Less expensive:

The screenshot shows a window titled 'Cluster Filter' with a list of 17 features, each preceded by an unchecked checkbox. The features are: TotalEnergy, AverageEnergy, MaxEnergy, PixelCount, Convexity, Width, CrosspointCount, VertexCount, RelativeHaloSize, BranchCount, StdOfEnergy, StdOfArrival, RelLowEnergyPixels, Class, ClFile, PxFile, ClIndex, and Branches. At the bottom of the list is a button labeled 'Check All Attributes'.

Figure 3.22 Check boxes with cluster features in the Cluster Filter application

- Width,
  - Convexity,
  - RelativeHaloSize.
3. Inexpensive:
- TotalEnergy, MaxEnergy, AverageEnergy,
  - VertexCount, CrosspointCount, RelLowEnergyPixels,
  - StdOfArrival, StdOfEnergy.
4. Cheap:
- Class, PxFile, ClFile,
  - PixelCount, ClIndex.

Note: The complexity factor should be considered only if the input data are large.

### ***Distribution selection***

Before processing, we need to select the distribution of samples in the output. For that, we set the last parameters:

- ***Class distribution*** (**even** / **proportional**): When choosing **even**, each of the classes will have the probability  $p(class) = \frac{1}{\text{Number of classes}}$  to be the next class in the output samples. If the option **proportional** is chosen, each class will have the probability of being the next to process proportional to its count in the input samples:

$$p(class) = \frac{\sum_{partition \in class} |partition|}{\sum_{anyclass \in classes} \sum_{partition \in anyclass} |partition|}$$

where  $|partition|$  is the number of samples (clusters) its **cl** file.

- ***Partition processing order*** (**serial** / **parallel**): Let  $P_1, P_2, \dots, P_n$  be a list of partitions containing samples of some class in the order they were listed in the input selection table. After selecting **serial**, the samples for the class will be selected serially from  $P_1$ , then from  $P_2$  and so on until from  $P_n$ . When parallel partition processing order is selected, the first sample is selected from  $P_1$  and  $i$ -th sample is selected from  $P_{(i \bmod n)+1}$ . Note: This

order is only the order of samples **inside** the class, keep in mind that each class is distributed by the previous parameter – Class distribution.

- **Ending condition** (first class / last class / first partition): This condition determines when the processing should terminate. By choosing the **first class** option, the algorithm ends when all the samples of any of the classes are fully processed. When **last class** is selected, the program selects samples until all samples for all classes are processed before terminating. If the user chooses the **first partition** option, the program finishes as soon as all samples from any of the class partitions are outputted.
- **Align class**: When we set the optional parameter – align class – to the name of some of the classes in the loaded files, then the termination of the algorithm is only based on partitions of this align class and on the ending condition. All the other classes are meanwhile processed but with the following adjustment. When a partition of the non-align class is fully processed, it is not removed from the list, but it is only reset and will be processed once again from the beginning. This means only the align class is being depleted while the others are repeatedly looping over their data.

Even though the first three parameters might actually seem useful at first glance, one might wonder about the usefulness of the align class parameter. When we want to achieve even class distribution during training, we have two possible options:

- Undersampling [11] – reducing the size of the majority class, and
- Oversampling – reusing the minority class more times.

One can notice that when choosing even class distribution (with an ending condition on the first-class) and no align class, this is an example of undersampling. In contrast, if the align class is set, that can be considered an implementation of the oversampling for classes with a smaller extent than the align class and simultaneously as undersampling for classes with a greater extent than the align class. After all these necessary steps are completed, the user can click the **Process** button, which starts the processing and writing the output in a JSON text file, see Figure 3.23.



```

{
  "id":1,
  "TotalEnergy":10032.440181,
  "AverageEnergy":51.4484111846154,
  "MaxEnergy":521.635851,
  "PixelCount":195,
  "Convexity":0.951219512195122,
  "Width":19.9248588451713,
  "CrosspointCount":0,
  "VertexCount":2,
  "RelativeHaloSize":0.871794871794872,
  "BranchCount":2,
  "StdOfEnergy":98.470644311926,
  "StdOfArrival":27.8158281889533,
  "RelLowEnergyPixels":0.333333333333333,
  "Class":"<none>",
  "ClFile":"../../../../Celko%202020%20Example%20data/data/particles/frag/f_nuc_cl.txt",
  "PxFile":"../../../../Celko%202020%20Example%20data/data/particles/frag/f_nuc_px.txt",
  "ClIndex":1
},

```

Figure 3.23 An example of calculated features for a single cluster

### 3.7 Classifier applications

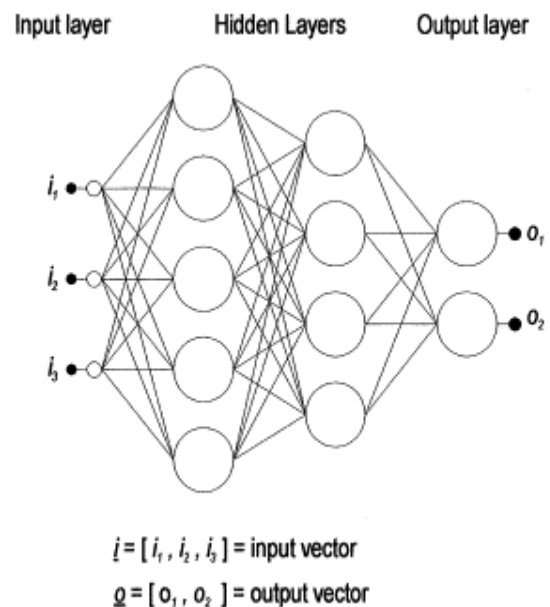
When we have the training data prepared by the

Description Generator application, we can start focusing on developing the tools for cluster classification.

For this purpose, we designed three applications:

- ClassifierUI,
- ClassifierTrainer, and
- ClassifierForClusters.

Each of these applications uses the same type of neural network classifier model – a Multi-layered perceptron (MLP) [12]. A multi-layered perceptron is a kind of neural network. Its architecture is shown in Figure 3.24. It consists of multiple layers, where each layer contains some non-zero number of neurons. The first layer of neurons is called the input layer, and the last layer is known as the output layer



(the layers in between are called

Figure 3.24 MLP architecture – circles represent neurons, lines correspond to the edges between them [12]

hidden layers). Each neuron is connected with all neurons in the previous layer and the next layer via edges (an edge from neuron  $i$  in a layer to neuron  $j$  in the next level is assigned some real number – weight  $w_{i,j}$ ). During the classification, we set the values of the neurons in the input layers and start updating neurons in the hidden layers from left to right. A single update operation consists of setting the neuron  $n_{i,j}$  ( $j$ -th neuron in the  $i$ -th layer  $l_i$ ) as follows:  $n_{i,j} = f(\sum_{j=1}^{|l_{i-1}|} w_{i,j}n)$  where  $f$  is known as the activation function. The output layer is updated as the last, and from this layer, the output of the whole network is retrieved.

This type of model has many learning parameters that can affect how quickly the model learns and how robust the model is (as displayed in Table 3.3).

<i>NN attributes and learning parameters</i>	
<i>name</i>	[any string] – used as a unique classifier identifier
<i>validAttributes</i>	[array of ClusterAttribute object] – sets the attributes of a cluster used during the learning process
<i>layerSizes</i>	[array of positive integers] – represents the sizes of the hidden layers
<i>activationFunction</i>	["relu", "sigmoid"] – sets one of the two currently supported activation functions
<i>learningAlgorithm</i>	["backProp", "leven-marq"] – chooses either the Backpropagation or Levenberg-Marquardt learning algorithm
<i>epochSize</i>	[positive integer] – the number of samples used in a single training epoch, known by the term batch size
<i>learningRate*</i>	[real number between 0 and 1] – the base step size in the gradient descent learning algorithm [13] (lower values = slow learning, higher values = possibly no convergence)
<i>momentum*</i>	[real number between 0 and 1] – a learning parameter that aims to speed up the convergence by setting the current step as a linear combination of the gradient and

	the previous step
<i>evaluationMultiClasses</i> *	[array of pairs ([string], [array of strings])] – maps the prediction classes in the array (the second element of the pair) to a new class name ( the first element of the pair) for the evaluation process (for example, if our classifier predicts classes <i>A</i> , <i>B</i> and <i>C</i> , we might want to analyze how well the classifier did at splitting <i>A</i> class from the rest (using the accuracy)), so we map [( <i>A</i> ,[ <i>A</i> ]),( <i>BC</i> , [ <i>B</i> , <i>C</i> ])
<i>usedTrainDataSize</i> *	[a real number between 0 and 1] – the proportion of data used for training (the remaining data are used for testing)
<i>printInterval</i> *	[positive integer] – the frequency of printing the training error of the neural network to the console during learning

**Table 3.3 Attributes and learning parameters of the classifier model stored in a config file (json),**

**\* = field is not obligatory**

*EvaluationMultiClasses* can be considered a non-standard parameter, so we present an example of how it might be useful. Suppose we want to train a classifier that gets the classes *A*, *B*, and *C* on the input. Its role is to separate class *A* from the class  $B \cup C$ . Then, when we are generating the training dataset, we might want to preserve even distribution between the classes *A*, *B*, and *C*. Hence, we create the training data that include all these three classes and train a classifier on those (the classifier will use all of the classes *A*, *B* and *C*). However, when we are evaluating this classifier, we should be more interested in the "quality" of the split between the classes *A* and  $B \cup C$ , rather than taking the split between *B* and *C* into account. For that specific case, we use the *EvaluationMultiClasses* parameter.

### ***Classifier UI and Classifier Trainer***

The **ClassifierUI** is an application that aims to provide two main functionalities – classifier training and classifier merging. **ClassifierTrainer** is a console application that also provides classifier training (the same as the **ClassifierUI**) but offers the option to run the learning process from the script

without manual interaction. Because the training process is the same in the console application as in the UI, we will not discuss it separately.

**Classifier Training** – The application trains a new classifier model based on the training data and the learning parameters provided by the user. Some parameters like the minimal accuracy can be set directly via the UI, but most of the learning parameters are set in a JSON config file, which must be loaded into the UI before learning starts. Optionally we can also load an already trained classifier model, and in this case, the training starts with this model instead of creating a new MLP by a random initialization of the neural network. By clicking **Train Classifier**, we start the learning process. Then, the data is randomly split into training and test set according to the learning parameter (by default, 90% are used for training, 10% for testing). Information about the error rate on the currently processed epoch is displayed in the console window. When the training is completed, the model is tested on the test data. To present the quality of the classifier, we used the confusion matrix [14] – a matrix  $C$  where its element  $C_{i,j}$  is the number of clusters that belong to the  $i$ -th class and were labeled by the classifier as the  $j$ -th class. So ideally, we want to have the most cluster examples on the diagonal of the matrix. The result of the training process is what we call a **simple classifier**, and it is stored in two files with the suffixes **.csf** and **.csf\_support**. We decided to use two files because of the problems with storing all of the classifier data into a single file (a more detailed description of the problem is provided in Chapter 4.3). These two files should be in the same directory and preserve the same name at all times (they only differ in suffixes).

**Classifier cascading** – When we have multiple simple classifiers, we might want to concatenate them into a single classifier with a tree-like structure. This can be especially useful when building a classifier for unbalanced classes. Instead of training one complex classifier, we can split the task into building several simple classifiers and then combine them into a complete classifier. Such modular classifier has additional good properties. We can concentrate on training simple classifiers for the most important classes, each of the simple classifiers can be retrained separately for the other simple classifiers, and its training can be much faster than training a large monolithic classifier. Let, e.g., classes  $A$  and  $B$  are rare while classes  $C$  and  $D$

are more abundant in our data. Because when we have classes with little training data (Classes *A* and *B*) as displayed in Figure 3.25, we can create a root classifier trained on an evenly distributed dataset by undersampling the larger classes (Class *C* and *D*). However, undersampling means not using some part of the training data, which could negatively affect the model's accuracy. Because it is possible to have more than one level, we can train another classifier on the larger dataset (of Classes *C* and *D*), which enables us to:

- use most of the training data for learning, and
- preserve the even distribution of classes among each level of the tree, which is preferred when training many machine-learning based classifiers

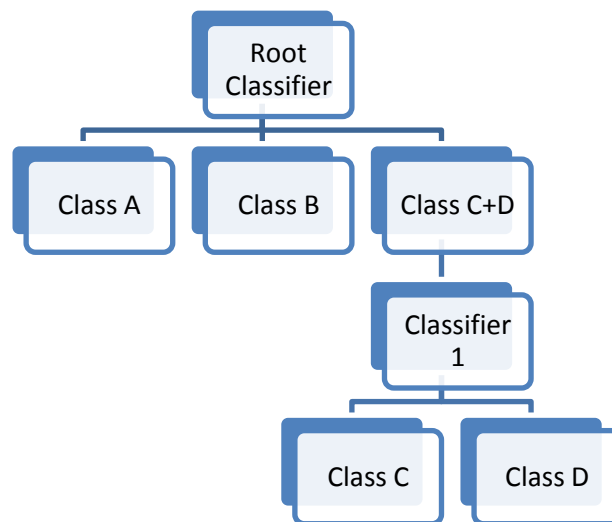


Figure 3.25 Example of a multi-layered classifier with two layers (Root classifier and Classifier 1)

Via the UI, it is possible to create a cascade classifier with a maximum of four levels. On each level, there can be a single classifier at most. This is because it can be quite complex to build an arbitrary tree via the UI, and also, we expect this version to be sufficient for most cases. With a little programming, the user can build any classifier tree by simply connecting the nodes.

To merge classifiers into a cascade classifier, the user imports the trained model, and if it is not the final layer, the user also specifies the `split class`, which is the name of the class that the classifier on the next layer will further process. This means that if we input  $n$  simple classifiers for merging, we should also input  $n - 1$  corresponding split classes. Then the name of the output is selected, and when the cascading is completed, two output files are created - `.csf` and `.csf_support` files.

## ***Classifier For Clusters***

The `ClassifierForClusters` is a console application. It can classify the given collection of clusters and produce an output based on its parameters when it is provided a trained classifier model and the cluster data in JSON format. There are three possible types of output:

- ***Frequencies:*** The application prints the key-value pairs of the name of the class and its frequency in the data.
- ***Classes:*** The program outputs a single JSON file for each of the classes, which contains the clusters that were predicted to belong to the particular category. When calculating the expected class, we apply the softmax function to the output layer of the network to convert the neuron values into probabilities of each class. In case we have  $K$  output neurons (classes), the output is computed as

$$\text{softmax}: \mathbb{R}^K \rightarrow [0,1]^K \text{ and } \text{softmax}(x)_i = \frac{e^{x_i}}{\sum_{j=1}^K e^{x_j}}$$

The cluster is unclassified if the most probable class is assigned a similar probability as the second most probable class.

- ***Specials:*** When this option is selected, the program's output is similar to the Classes option, but additionally, it outputs a file where all the unclassified clusters that match specific criteria (potentially rare clusters) are stored. These criteria select the clusters with more than four branches and at least 100 pixels. Changing the criteria is simple but requires a slight modification of the source code.

## 4 Development documentation

In this chapter, we will discuss the technical part of the solution ClusterProcessor. We will have a brief look into the implementation and the main objects and methods used in the solution. This section is primarily aimed at the developers who would like to find out how our solution works and those who plan to extend this solution in any way.

### *Technology*

The framework we decided to use for the solution is .NET Framework and Windows Forms. The reason for that is simply because my primary programming language is C#, and I consider Windows Forms to be a reasonable choice for creating Graphical User Interface. However, WinForms does not support all features we required, so we need the following external libraries:

- Chart Director [15] – a 3D graphical plotting library,
- Json.Net [16] – a library (NuGet package) used for serialization and deserialization from the JSON file format, and
- Accord.Net [17] – a machine-learning framework (NuGet package) we use to classify the clusters.

### *Solution structure*

The whole solution represents a set of tools for cluster processing. The solution is structured into multiple separate projects mostly for two reasons:

1. Users can choose which parts of the solution they need and use only those.
2. This way the whole solution can be easily extended by modifying only one project and reusing the rest of the solution. The projects in the solution can be divided into three layers according to their dependencies, as shown in Figure 4.1.

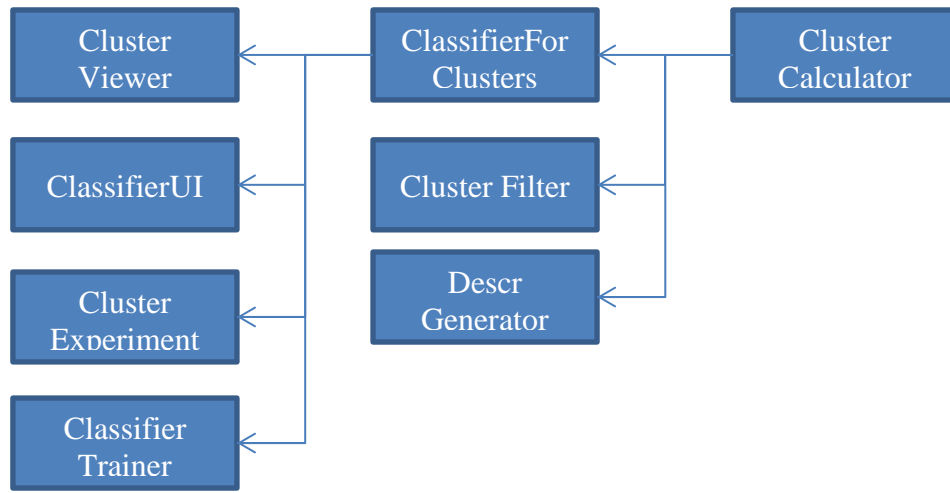


Figure 4.1 Dependency structure of the ClusterProcessor solution

## 4.1 Cluster Viewer

Even though the whole cluster viewer project consists mainly of the event handlers for the application's buttons, it uses some features from the `ClusterCalculator`, which are worth noting from the software engineering perspective.

### *Browsing Cluster Collection*

Browsing of the collection of clusters is controlled via buttons. Because the size of the dataset a user might be browsing can be huge (millions of clusters), we do not have a chance to load the data into operating memory, but we need to search for a specific cluster sequentially each time we click `previous`. There is an option to use caching of the recently viewed clusters. We tested the browsing on the files with large `c1` files (roughly ten million clusters) and did not notice any significant issue (sequential browsing is done only on the `c1` file, which is usually significantly smaller than the `px` file). Because of that, we decided not to try optimizing browsing any further. Browsing is implemented by `ClusterInfoCollection : IEnumerable`.



## ***Skeletonization***

Skeletonization is the process of finding the skeleton of a binary image. For this purpose, we proposed Algorithm 3.1. As the thinning process returns non-trivial data for bigger clusters, we optimized the algorithm for clusters with more than 100 pixels. This is because the asymptotic complexity is only relevant for bigger clusters. The key is to use a well-fitting data structure. Throughout the algorithm, we perform two operations most often. The first one is `.Contains()`, which means 'does the specific pixel have a non-zero energy ?' and the second one is `.Delete()` which is setting the energy of a specific pixel to 0. An ideal option for these operations seems to be an array of  $256 \times 256 = 65\,536$  pixels, which can effectively do both these operations in constant time. The problem is that this approach would lead to huge performance issues as most of the clusters are smaller than 100 pixels, and the overhead of the array with 65 536 items would take much time. Using the list collection `List<T>` from the standard library, we get rid of this problem. On the other hand `.Contains()` and `.Delete()` require time linear with respect to the length of the list. That is why we chose to use `HashSet<T>`, which has similar memory usage as `List<T>` but both `.Contains()` and `.Delete()` are done in a constant time. In case we would need another performance boost, the whole algorithm can be almost trivially parallelized.

## **4.2 Filter and Description Generator**

Both `ClusterFilter` and `DescriptionGenerator` calculate the features for many clusters, so we decided to analyze them together. The applications can run on a specific collection for a potentially long time, so performance seems to be a good factor we should focus on and optimize.

### ***Complexity***

The iteration process over the cluster collection uses `ClusterInfoCollection`, and as we need to process every item in the

collection, there is not much to be optimized. Let  $p$  denote the size of the `px` file and  $c$  the size of the `cl` file. The iteration process has a linear time complexity with respect to  $c$ . The calculation of an attribute value depends on the particular attribute. Asymptotic time complexities with respect to the number  $n$  of pixels of a cluster are shown in Table 4.1.

ATTRIBUTE	ASYMPTOTIC TIME COMPLEXITY
PixelCount, ClIndex, PxFile, ClFile, Class	$O(1)$ : These attributes are present when loading the cluster – no computational overhead.
TotalEnergy, MaxEnergy, AverageEnergy, RelLowEnergyPixels, StdOfEnergy, StdOfArrival	$O(n)$ : Calculating these properties only requires an iteration over each pixel in the cluster. Therefore, we obtain the linear complexity.
Width, RelativeHaloSize, Convexity, VertexCount, CrossPointCount	$O(nt + n \log n)$ : Because all of these attributes require skeletonization of the cluster, we need to analyze Algorithm 3.1. Each skeletonization iteration consists of iterating over $O(n)$ pixels and there are $t$ iterations, where $t$ is the thickness of the cluster. (Note: Convexity and Width attribute also require convex hull calculation, which is done in $O(n \log n)$ , which affects the asymptotic complexity)
BranchCount, Branches	<p><math>O(n^2)</math> originally, <math>O(n)</math> after update: When we look at Algorithm 3.2, we can notice the following:</p> <ul style="list-style-type: none"> <li>• The time complexity of the breadth-first search [18] is linear with respect to the number of pixels.</li> <li>• The branch count could be at most linear with respect to the number of pixels.</li> </ul> <p>Thus, we can estimate the worst-case complexity by <math>O(n^2)</math>. In contrast, we usually do not need to</p>

	<p>analyze the cluster much further if we already have hundreds of its longest branches. Because of that, we set the upper bound for branch count to be 40. We also set the depth of the calculation, so the algorithm ignores the branches that are nested more than four times. Even though this restriction doesn't provide us a complete branch analysis, in most cases, it is sufficient and provides a significant performance boost to <math>O(n)</math>. (Note: From the real (non-asymptotic) observed calculation time, it is still the most expensive calculation.)</p>
--	--

**Table 4.1** Time complexity of the feature calculation

To sum up, the total complexity of the filtering process is  $O(\sum_{k=1}^c n_k t_k + n_k \log n_k)$  where  $n_k$  is the number of pixels and  $t_k$  is the thickness of  $k$ -th cluster (when taking the updated branch analysis into account).

### ***Adding new features***

To add a new feature to `DescriptionGenerator`, a user needs to follow these steps:

1. Add the name of the attribute to the enum `ClusterAttribute` to the `ClusterCalculator` project.
2. Add a case to the switch section of the `CalcAttributes` method of the `DefaultAttributeCalculator` with your attribute and its calculation and store the value to the dictionary `attributePairs` to an index `attribute` (`attributePairs[attribute] = calculated_value`).
3. In the `DescriptionGenerator` project, update the UI – add a checkbox to the other attributes with the same name as the enum you added in step 1 (because the string-enum conversions are done automatically).

## 4.3 Classifier applications

In this chapter, we discuss the process of cluster classification from a developer's point of view. We introduce some fundamental objects used in the classification process. Then we also mention a simple guide on how to build your own classifier.

### *Classifier objects*

**Interface IClassifier** is an interface that provides required methods for a classifier model (e.g., `.Classify()`, `.StoreToFile()`, `.LoadFromFile()`)

**Interface ILearnableClassifier : IClassifier** represents an interface that extends the `IClassifier` by adding support for training (`.Train()` method)

**Class NNClassifier : ILearnableClassifier** is a basic classifier based on a multi-layered perceptron NN,

**Class MultiLevelClassifier : IClassifier** represents a classifier created by cascading more `NNClassifiers`,

**Class ClassPrediction** – an object that represents the prediction calculated by the classifier, apart from the name of the predicted class, also contains the probabilities of how confident the classifier is about each class (implemented via the softmax function).

### *Building your tree classifier*

When you want to build a `MultiLevelClassifier` you first need to create a `NNClassifier`, which can be done two ways:

- a) from the `ClassifierUI`, or
- b) using a console application `ClassifierTrainer`.

After we have all of the `NNClassifiers` prepared, we can start with merging. There is a possibility to cascade classifiers in the `ClassifierUI`, but it only allows a single classifier on each level. Still, it is possible to build any classifier tree, by the following steps (an example of how we the classifiers are cascaded in the

current implementation can be seen in the method `.FromDefault()` of the class `MultiLevelClassifier`):

1. Create new instances of `NNClassifier` and call `.LoadFromFile()` and create a new instance of `MultiLevelClassifier`.
2. Wrap the simple classifiers into the class `ClassifierNode` and the root classifier into the `ClassifierRoot`.
3. Connect the nodes, starting from the bottom of the tree, by calling `parentClassifier.Descendants.Add(splitClass, sonClassifier)`.
4. After all of the previous steps are done, set the root by calling the method of the `MultiLevelClassifier` - `SetRoot(classifierRoot)`

### ***Serialization***

Each classifier that is created can also be stored in a file for future use. This process is called serialization and is done using the Newtonsoft library. When we were implementing the serialization of both `NNClassifier` and `MultiLayeredClassifier`, we needed to set all of the property modifiers to public so they can be visible for the serializer. Another problem is that the network itself (from the Accord.Net library) cannot be serialized using the Newtonsoft library because of compatibility issues. That is the main reason why as a result, each classifier is stored in two files (`.csf` and `.csf_support`). The `.csf` file contains a serialized Network by the serialization tools of the Accord.Net library. In contrast, the `.csf_support` file contains the whole `NNClassifier` object except for the network itself (serialized by the Newtonsoft library).

### ***Selecting uncommon clusters***

One of the ultimate goals of the classification could be finding uncommon clusters. The selection of such clusters is implemented directly in the classification process. `ClassPrediction` contains the method `.CalcConfidence()`, which calculates the certainty of the prediction based on the output neuron values from the

network. This method serves to detect unclassified clusters. Let  $p_{max}$  denote maximal probability we obtain from applying softmax to the output layer and let  $p_{max2}$  be the second maximal probability. We consider a cluster to be unclassified if the difference  $p_{max} - p_{max2} < \varepsilon$ , where we set  $\varepsilon = 0.05$ . During the classification, the method `.CheckSpecialClusters()` is called, which takes the unclassified cluster and decides if it can be considered an extraordinary cluster (it checks if it is unclassified and has more than 100 pixels and more than 3 branches). Both `.CalcConfidence()` and `.CheckSpecialClusters()` are marked as virtual so that they could be potentially reimplemented simply by inheritance. This is also where we see a chance for improvement and extension of the program.

## 5 Experiments

During the work on the thesis, we created several classifier models tested them on real datasets. For that, we designed a couple of scenarios where we compare our approach with the alternatives and analyze the results. In all experiments, we will classify the clusters in the same base set of classes (displayed in

Table 1.3): electron, muon, pion, proton, helium, fragment, iron, low energy electron, and lead. Apart from the mentioned classes, we also added a new artificial class **elPi0**. This class represents electrons and muons that traversed the detector perpendicularly, which we believe are extremely difficult to distinguish because they both only consist of very few pixels and have similar energy.

To perform all of the experiments, we prepared the following single-layered types of models:

1.  $S_{lead}$  – classifies the particles into lead particles and the rest. This classifier was trained on roughly 80000 clusters.
2.  $S_{fr\_he\_fe}$  – separates fragments, helium, and iron from each other and the rest. The dataset for this classifier consists of 450000 clusters.
3.  $S_{le\_pr}$  – separates low-energy electrons from protons and the rest. The total number of clusters we used during the training is more than one million.
4.  $S_{e\_m\_p\_ep}$  – splits clusters into four classes – electrons, muons, pions, and the artificial class **elPi0**. It is trained on the dataset of approximately 500000 clusters.
5.  $S_{all}$  – splits clusters into all categories directly (including the artificial category **elPi0**) and is trained on roughly 100000 clusters.

During the training of the mentioned classifiers, we used the values of parameters as shown in Table 5.1. In all classifiers except for  $S_{fr\_he\_fe}$  we only used a single iteration over the training set. The classifier  $S_{fr\_he\_fe}$  required 3 iterations to converge.

<b>validAttributes</b>	TotalEnergy, AverageEnergy, MaxEnergy, PixelCount, Convexity, Width, CrosspointCount, VertexCount, RelativeHaloSize, BranchCount, StdOfEnergy, StdOfArrival and RelLowEnergyPixels
<b>layerSizes</b>	[13, 13]
<b>learningAlgorithm</b>	backProp
<b>epochSize</b>	8
<b>learningRate</b>	0.6
<b>momentum</b>	0.5
<b>activationFunction</b>	sigmoid

Table 5.1 Base model parameters of  $S_{lead}$ ,  $S_{fr\_he\_fe}$ ,  $S_{le\_pr}$ ,  $S_{e\_m\_p\_ep}$  and  $S_{all}$  classifiers

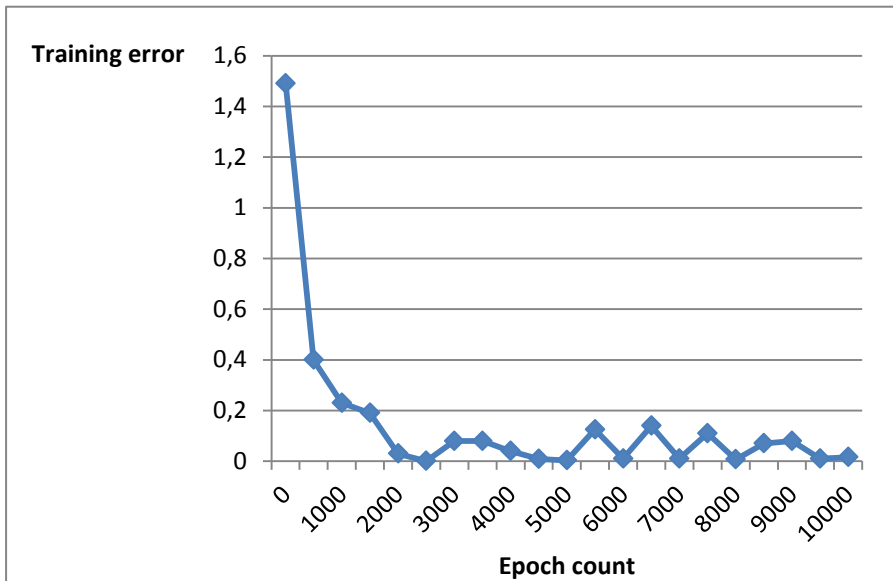


Figure 5.1 Training mean square error with respect to epoch count

For each classifier we trained in the experiments, we made sure the learning process was completed – this was done by observing the training error during the learning. If the training error of the classifier stops decreasing, the training can be stopped. As an example, we plot the error of a single classifier during the training, which is displayed in Figure 5.1.



Using the single-layered classifiers and given training data distribution (the clusters with similar frequency in the training dataset are on the same level), we also constructed the multi-layered classifier  $M_{all}$  as displayed in Figure 5.2.

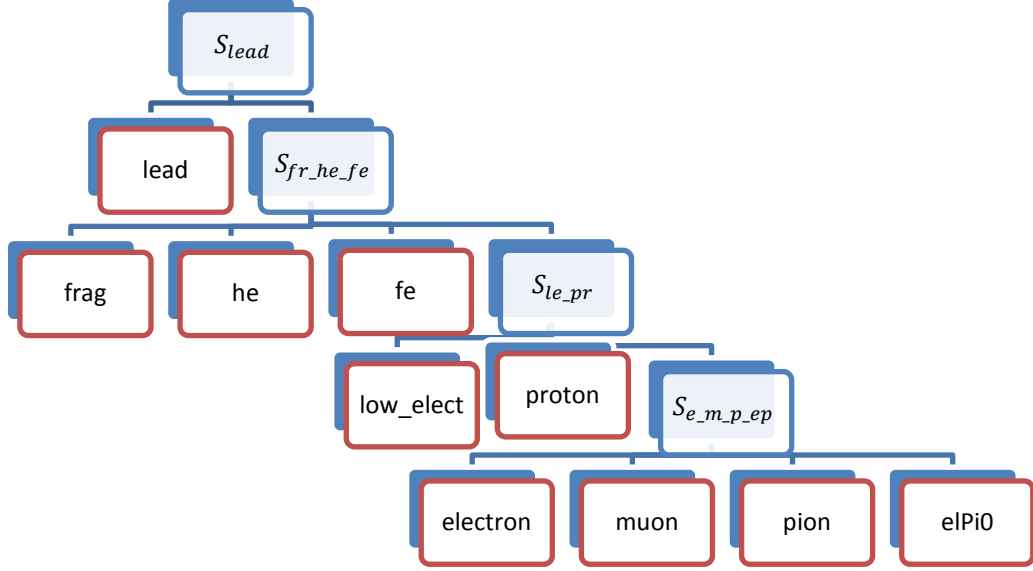


Figure 5.2 Multi-level classifier  $M_{all}$  structure

In order to measure the quality of a classifier, we define the **accuracy** of a classifier  $C$  on the dataset  $D$  as  $Acc_D(C) = \frac{\text{number of correctly classified examples}}{\text{number of all classified examples}}$ .

## 5.1 Classifier Parametrization

Each single-layered classifier trained via our solution has multiple parameters, as shown in Table 3.3. In this experiment, we modified the values of some of these parameters and compared the results. As a base model for this experiment, we used the classifier  $S_{fr\_he\_fe}$ .

Because there are many possible parameters to modify and training of multiple classifiers on large datasets is time-consuming, we chose to modify only a few parameters:

- learning rate
- momentum, and
- the number of layers of the neural network

To demonstrate the significance of the learning parameters, we decided to try six scenarios (every scenario was tested on ten newly trained classifiers  $S_{fr\_he\_fe}$ ), where we used the same test dataset for each scenario:

**1. *Low learning rate and momentum***

- learning rate: 0.1, and
- momentum: 0.1.

Result: average normalized test accuracy of the model: 0.86.

Interpretation: First, we need to note that the dataset we use for training this network is large (almost 500 000 clusters). This means that despite low values of learning rate and momentum (slow learning), the classifier had enough time to converge to a reasonable accuracy.

**2. *Medium learning rate and momentum***

- learning rate: 0.5, and
- momentum: 0.5.

Result: average normalized test accuracy: 0.90

Interpretation: These values of parameters showed the best convergence speed and accuracy. This is the reason why we use these values as the defaults in our experiments.

**3. *High learning rate and momentum***

- learning rate: 1, and
- momentum: 1.

Result: average normalized test accuracy: 0

Interpretation: When we set the values of learning parameters too high, we can notice that a divergence occurs, which causes the classifier to produce unusable results. In our case, it caused all predictions to be unclassified.

#### **4. *Small number of hidden layers***

- A single hidden layer with one neuron.

Result: an average normalized test accuracy: 0.67

Interpretation: Using a smaller network usually implies faster learning, but tiny networks do not perform very well on more complex problems.

#### **5. *Medium number of hidden layers***

- Two hidden layers with 13 neurons each.

Result: an average normalized test accuracy: 0.90

Interpretation: This configuration seems to provide a classifier with the best accuracy for our task. We can observe that the network is large enough to capture the various relationships of the features and small enough to learn reasonably fast.

#### **6. *Higher number of hidden layers***

- Three hidden layers with 13 neurons each.

Result: an average normalized test accuracy: 0.25

Interpretation: We can notice that the accuracy is low. This could be because the network is too deep for our task and our chosen activation function (sigmoid function [19]), and therefore, it learns very slowly.

To sum up, the results of the training depend on the values of learning parameters. Changing just a single parameter of the network to an extremal value can cause the learning algorithm not to converge, so one needs to be careful when setting the values of parameters. There are some recommendations on how the parameters can be set to achieve convergence in most of the cases:

- The more data we have for training, the deeper neural network we might need to use.
- First, we start with low values of learning rate to prevent divergence, and then we can try to increase it to reach convergence faster.
- When we are using the sigmoid activation function with backpropagation, we can try networks with only a few layers because training a deeper

network can be very slow with this algorithm. In contrast, when using rectified linear unit [20] activation function, we can also try deeper architectures of neural networks.

However, there is no general rule on how to set the parameters to get the best accuracy. For our specific task, the parameters we used were the best we came up with, but we do not claim that these parameters are the best possible. Similarly, each classification problem might require different values of training parameters to get the optimal accuracy.

## 5.2 K-Fold cross-validation of single-layered models

Because the training of the classifier is, in general, a non-deterministic procedure, we should also test the training method and how often the classifier training is successful. Another property we might be interested in is the variance of these results, which was investigated in the following experiment. One of the most commonly used methods for training evaluation is the  $k$ -fold cross-validation [21]. There, the data is randomly partitioned into  $k$  folds and the classifier is trained  $k$  times. In the  $i$ -th validation run ( $i = 1, \dots, k$ ), the  $i$ -th fold is used for testing after training the classifier on the data from the remaining  $k - 1$  folds. We performed 6-fold cross-validation on the dataset and obtained results (accuracies) as shown in Table 5.2.

Val_run\classifier	$S_{lead}$	$S_{fr\_he\_fe}$	$S_{le\_pr}$	$S_{el\_mu\_pi\_elPi}$
1. fold	0.987	0.921	0.951	0.705
2. fold	0.979	0.922	0.950	0.747
3. fold	0.981	0.912	0.957	0.733
4. fold	0.990	0.938	0.938	0.740
5. fold	0.977	0.938	0.943	0.741
6. fold	0.986	0.900	0.959	0.740
Mean	0.983	0.922	0.950	0.734
Std deviation	0.002	0.027	0.001	0.006

Table 5.2 Results of the cross-validation of a single-layered model

The results of the 6-fold cross-validation indicate that our classifiers have a stable performance (relatively low standard deviation). We can also notice that the categories electron/muon/pion have lower accuracies which could mean that these categories are challenging, if not impossible, to classify correctly with high accuracy.

### 5.3 Multi-level classifier

The concept of a multi-level classifier is something we use in this thesis frequently. But how does it perform in comparison with a single-level classifier on the real data?

In order to compare these two classifiers, we trained ten single-level  $S_{all}$  models and ten multi-level  $M_{all}$  classifiers on a similar amount of data and compared the results on the test dataset.

The best  $S_{all}$  model accuracy was 0.61 (mean 0.52), while the best accuracy of the  $M_{all}$  model reached 0.82 (mean 0.80). We also estimated the variance of  $S_{all}$  to be 0.0038 while the variance of  $M_{all}$  was only 0.0001. Confusion matrices of these models can be seen in

Table 5.3 and

Table 5.4. In the tables, it is visible that the particle distribution in the test set is very uneven. However, this is not a problem because we can normalize each value by

dividing it by the sum of its row. After rerunning the experiment and computing the normalized confusion matrix, we obtain the best relative accuracy of 0.85 for  $M_{all}$  model and 0.59 for  $S_{all}$  model.

Based on the results of the experiment, we conclude that the accuracy of the multi-level classifier was significantly higher than the single-level model with both classifiers using similar architectures of the neural networks. One reason why there is such a difference could be the fact that multi-layered models do not suffer from undersampling (and oversampling) as much as single-level models. Another reason could be the scaling – each single-level classifier scales its inputs (linear Min-max scaling [22]) into the range (0,1), which means having a type of a cluster with very high energy causes the scaling to scale most of the other cluster energies very close to 0, which makes them difficult to separate. In contrast, the multi-level classifier we built first processes the bigger cluster types with higher energy (first layer) and then proceeds to smaller cluster types with less energy which causes the scaled values to be more evenly spread in the interval (0,1). An important note – by no means we state that the multi-level architecture of the classifier is always better than single-level architecture. It might depend on the type of the classifier and the training data, but in our case, the differences were significant.

True\Pred	lead	frag	he	fe	prot	low_e	elPi0	pion	muon	electr	uncl
lead	283	0	0	9	0	0	0	0	0	0	0
frag	17	362	31	200	64	0	0	0	0	0	10
he	1	1537	2321	4392	830	3	17	0	0	1	206
fe	3	2	0	1769	0	0	0	0	0	0	0
prot	0	550	126	94	31011	152	2860	67	408	414	315
low_e	0	0	0	0	83	1525	1291	28	29	37	7
elPi0	0	20	0	0	721	43	4640	78	130	278	221
pion	3	10	0	0	326	28	1233	4106	238	2835	90
muon	11	102	11	0	678	11	624	30	1303	126	104
electr	46	390	0	0	2806	80	3301	1235	2527	4167	448
uncl	0	0	0	0	0	0	0	0	0	0	0

**Table 5.3** Confusion matrix of the model  $S_{all}$  with the best accuracy in the experiment (uncl represents unclassified clusters)

True\Pred	lead	frag	he	fe	prot	low_e	elPi0	pion	muon	electr	uncl
lead	287	0	0	5	0	0	0	0	0	0	0
frag	22	609	44	1	0	0	0	0	0	0	8
he	1	1295	7899	1	37	8	0	2	0	0	65
fe	2	9	0	1749	0	0	0	0	0	0	14
prot	1	173	1528	0	33111	236	139	102	426	207	74
low_e	0	0	0	0	36	2771	14	16	40	16	107
elPi0	0	0	0	0	176	51	7886	36	406	402	43
pion	0	0	3	0	55	186	37	5051	97	41	30
muon	0	0	0	0	71	38	29	112	2469	264	17
electr	0	0	0	0	681	180	3002	406	3543	7050	138
uncl	0	0	0	0	0	0	0	0	0	0	0

Table 5.4 Confusion matrix of the model  $M_{all}$  with the best accuracy in the experiment (uncl represents unclassified clusters)

### *The best classifier*

Apart from the classifiers we trained for the experiments, we also repeated the process multiple times, iterating over the training set, and this way got a model with better results than the models trained in the experiments. When we ran this classifier on the test dataset and normalized the confusion matrix, we reached an accuracy of 0.875 (87.5%). The normalized confusion matrix for this classifier is displayed in Table 5.5. We also decided to save this model for future use - it is stored in the files `bestClassifier.csf` and `bestClassifier.csf_support`, which are a part of the presented solution.

True\Pred	lead	frag	he	fe	prot	low_e	elPi0	pion	muon	electr	uncl
lead	0.921	0.034	0	0.044	0	0	0	0	0	0	0
frag	0.004	0.888	0.102	0	0	0	0	0	0	0	0.004
he	0	0.054	0.94	0	0	0	0	0	0	0	0.002
fe	0	0.006	0	0.976	0	0	0	0	0	0	0.016
prot	0	0.004	0.022	0	0.933	0.006	0.006	0.01	0.004	0.009	0.001
low_e	0	0	0	0	0.005	0.957	0.012	0.011	0.004	0.008	0
elPi0	0	0	0.001	0	0.032	0.032	0.897	0.007	0.029	0.015	0.003
pion	0	0	0	0	0.007	0.006	0.002	0.821	0.044	0.109	0.007
muon	0	0	0	0	0.011	0.011	0.024	0.006	0.841	0.102	0.003
electr	0.003	0	0	0	0.016	0.011	0.026	0.163	0.198	0.571	0.008
uncl	0	0	0	0	0	0	0	0	0	0	0

**Table 5.5 Normalized confusion matrix of the best trained model**



## 6 Conclusion

The main focus of the thesis was to implement a set of tools for processing the clusters from the Timepix3 chip in the MM data format. We had to create an interactive tool for cluster visualization and non-interactive tools for cluster filtering, training data generation, and cluster classification with an option for training new classifiers according to the user's needs. We consider all of these goals to be successfully completed.

### Attribute calculation

As one of the essential tasks in our project was analyzing the clusters, we proposed and implemented several attributes of a cluster. These attributes were designed to provide a more compact representation of clusters and allow for a simpler distinction between various types of clusters. Calculation of some of these features relies on an image skeleton where we suggested a slight modification of the Zhan-Shuen's algorithm. Arguably the most interesting part of the feature calculation was the branch analysis, where we introduced a new algorithm based on the breadth-first-search.

### Visualization

We created a project for cluster visualization that provides 2D and 3D image representations of a cluster. Apart from that, the application also displays a histogram of the energy of the pixels in the currently viewed cluster as well as the histogram of the pixel count for the whole cluster collection. By using feature calculation, we are also able to show cluster attributes to the user. It is also possible to load a trained classifier into the viewer and use the classifier directly inside the viewer to predict the cluster's class.

### Classification

One of the most challenging tasks in the whole thesis was cluster classification. Firstly, we created a tool for feature-based training data generation in JSON format. A new type of multi-level classifier architecture was proposed. It consists of

connecting the results of multiple classifiers into a tree-like structure. Then, we implemented an application that is capable of training a simple neural network classifier based on the parameters selected by the user that could reflect the expected distribution of the cluster classes in a dataset. These classifiers can then be stacked into a multi-layered classifier. To present the functionality of the training application, we created a multi-level classifier model for classifying the real existing dataset of clusters.

### **Future research**

Because each tool is implemented by a separate application, it should not be difficult to extend our solution in many ways. As one of the most promising extensions, we see experimenting with different cluster features and examining their internal dependencies to provide the best possible accuracy in classification. Another important extension of our application could be the reimplementation of the function for selecting rare unclassified clusters based on the new features like the number of branches with length (or energy) above a given threshold. In the end, all of these steps could bring us closer to the ultimate goal – reliable detection of exotic particles.

## Bibliography

- [1] CERN – Timepix. [Online] [Cited: April 1, 2021.] <https://kt.cern/technologies/timepix3>.
- [2] The ATLAS detector. [Online] [Cited: April 4, 2021.] <https://atlas.cern/discover/detector>.
- [3] Meduna, Lukáš. Multifile-Clustering application. *IEAP Software Repository*. [Online] [Cited: September 10, 2020.] <https://software.utef.cvut.cz/>.
- [4] Medipix1. [Online] [Cited: March 15, 2021.] <https://medipix.web.cern.ch/medipix1>.
- [5] Llopart, X., et al. Medipix2, a 64k pixel read out chip with 55 /spl mu/m square elements working in single photon counting mode. *2001 IEEE Nuclear Science Symposium Conference Record (Cat. No.01CH37310)*. 2001, Vol. 3, pp. 1484-1488.
- [6] *A universal algorithm for image skeletonization and a review of thinning techniques*. Saeed, Khalid, et al. 2, s.l. : University of Zielona Góra, 2010, International Journal of Applied Mathematics and Computer Science, Vol. 20, pp. 317-335.
- [7] *3D track reconstruction capability of a silicon hybrid active pixel detector*. Bergmann, Benedikt, et al. 421, s.l. : Springer, 2017, The European Physics Journal C, Vol. 77. 9 pages. 6.
- [8] Silicon detectors. [Online] [Cited: April 21, 2021.] <https://indico.cern.ch/event/124392/contributions/1339904/attachments/74582/106976/IntroSilicon.pdf>.
- [9] Meduna, Lukáš. *Detecting elementary particles with Timepix3 detector*. Charles University. 2019. Master thesis.
- [10] *A Fast Parallel Algorithm for Thinning Digital Patterns*. T. Y. Zhang, C. Y. Suen. 3, s.l. : Association for Computing Machinery, 1984, Communications of ACM, Vol. 27.
- [11] *An Application of Oversampling, Undersampling, Bagging and Boosting in Handling Imbalanced Datasets*. Yap, B. W., et al. [ed.] T. Herawan, M. Deris and J. Abawajy. s.l. : Springer, 2014. Proceedings of the First International Conference on Advanced Data and Information Engineering (DaEng-2013). Vol. 285.
- [12] *Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences*. Gardner, M. W. and Dorling, S. R. 14-15, s.l. : Elsevier, 1998, Atmospheric Environment, Vol. 32.
- [13] Ruder, Sebastian. An overview of gradient descent optimization algorithms. [Online] [Cited: February 4, 2021.] <https://arxiv.org/pdf/1609.04747.pdf>.
- [14] Confusion Matrix. [Online] [Cited: March 28, 2021.] [https://en.wikipedia.org/wiki/Confusion\\_matrix](https://en.wikipedia.org/wiki/Confusion_matrix).
- [15] Chart Director library. [Online] Advanced Software Engineering Ltd. [Cited: October 17, 2020.] <https://www.advsofteng.com/download.html>.
- [16] Json .NET library. [Online] [Cited: April 4, 2021.] <https://www.newtonsoft.com/json>.
- [17] Accord .NET Framework. [Online] [Cited: January 17, 2021.] <http://accord-framework.net/>.

- [18] Bundy, Alan and Wallen, Lincoln. Breadth-First Search. *Catalogue of Artificial Intelligence Tools*. s.l. : Springer, 1984.
- [19] Han, J. and Moraga, C. The influence of the sigmoid function parameters on the speed of backpropagation learning. [ed.] F. Sandoval and J. Mira. *from Natural to Artificial Neural Computation*. 1995, Vol. 930, p. Springer.
- [20] Hara, K., Saito, D. a Shouno, H. Analysis of function of rectified linear unit used in deep learning. *2015 International Joint Conference on Neural Networks (IJCNN)*. 2015, s. 1-8.
- [21] *Cross-validation*. Berrar, D. s.l. : Elsevier, 2018, Encyclopedia of Bioinformatics and Computational Biology, Vol. 1, pp. 542-545.
- [22] Feature scaling. [Online] [Cited: February 27, 2021.] [https://en.wikipedia.org/wiki/Feature\\_scaling](https://en.wikipedia.org/wiki/Feature_scaling).
- [23] Chen, Yung-Sheng . The use of hidden deletable pixel detection to obtain bias-reduced skeletons in parallel thinning. *Proceedings of 13th International Conference on Pattern Recognition*. 1996, Vol. 2, pp. 91-95.
- [24] *A one-pass thinning algorithm and its parallel implementation*. Chin, Roland T., et al. 1, 1987, Computer Vision, Graphics, and Image Processing, Vol. 40, pp. 30-40.
- [25] *Fast fully parallel thinning algorithms*. Guo, Zicheng and Hall, Richard W. 3, s.l. : Elsevier, 1992, Computer Vision and Image Understanding, Vol. 55, pp. 317-328.

## List of Tables

Table 1.1 Input MM file format .....	6
Table 1.2 Categories of clusters based on their shape .....	10
Table 1.3 Types of the clusters for classification with their examples .....	12
Table 3.1 Description of functions and variables used in Algorithm 3.1. ....	26
Table 3.2 User interface of the filterer - setting the lower and upper bound for filters .....	32
Table 3.3 attributes and learning parameters of the classifier model stored in a config file (json), .....	38
Table 4.1 Time complexity of the feature calculation .....	46
Table 5.1 Base model parameters of Slead, Sfr_he_fe , Sle_pr, Se_m_p_ep and Sall classifiers.....	51
Table 5.2 Results of the cross-validation of a single-layered model .....	56
Table 5.3 Confusion matrix of the model Sall with the best accuracy in the experiment (uncl represents unclassified clusters).....	57
Table 5.4 Confusion matrix of the model Mall with the best accuracy in the experiment (uncl represents unclassified clusters).....	58
Table 5.5 Normalized confusion matrix of the best trained model.....	59

# Attachments

The attachments, which are part of a digital archive, are the following:

## **A. ClusterProcessor - user documentation**

ClusterProcessor user documentation consists of the user guide and tutorials with examples of how to use the applications that are part of the solution.

## **B. ClusterProcessor - solution**

ClusterProcessor solution is a Visual Studio 2019 solution that contains the source code of all the tools we implemented for cluster processing.

## **C. ClusterProcessor – executables**

This is a set of (compiled) Windows executable files with all implemented tools.

## **D. ClusterProcessor – demo data**

Demo data folder contains small sample data intended mainly for getting familiar with the implemented tools. Bigger data, including the data used for training the classifiers, can be found on the following link: [https://drive.google.com/file/d/1f\\_t0RE7KiqO69yrzWAbU8LSX8r3l4m\\_J/view?usp=sharing](https://drive.google.com/file/d/1f_t0RE7KiqO69yrzWAbU8LSX8r3l4m_J/view?usp=sharing).