JUSTUS-LIEBIG-
UNIVERSITÄT
GIESSEN
II. Physikalisches Institut, Gießen

Detecting Clusters in Highdimensional Data

Study Project

as part of the masters program in physics

by
Johannes Bilk and Johannes Budak
Johannes.Bilk@physik.uni-giessen.de, Johannes.Budak@physik.uni-giessen.de

Supervisor
PD Sören Lange

Gießen, August 2020

Abstract

In this work we talk about classifying clusters within data from the pixel detector of the Belle II experiment. Recent works on this [[1],[2]] using Self-organizing-maps were not able to differentiate more than one particle cluster from the background cluster, thus more straight forward and deterministic approaches using high dimensional voxels for covering the data clusters were examined. Two different methods are presented in this work. The first uses spherical voxels with stochastic elements. The second uses a regualar grid in the high-dimensional input space and traditional voxels. After covering the data with voxels, neighborhood relations are used to determine the voxels that form a cluster. We achieve classification efficiencies of up to 99% for test data sets with clearly separated clusters up to six dimensions. The methods used on the simulated pixel detector data are able to cover at least 70% with up to 100% of the data. However further modifications in the algorithms are needed to differentiate between clusters, that are not sufficiently separated from each other.

Contents

List of Figures

1 Introduction

In the field of high energy physics a lot of data is produced with hundreds and thousands of events.Because one does not know which events belong together, one cannot simply pick out the events from the thing they are searching for in the data. But this task gets worse, since every kind of measuring instrument introduces some kind of background, which thankfully is very characteristic.

One of the most common and essential methods of handling with that big amounts of data is data clustering. The main goal of data clustering is to separate between different kind of entities based on their different features represented by the data [6]. The objective of the data clustering process is to separate the data set into groups which are as similar as possible in their features in the same group, while they are as different as possible from the other groups.

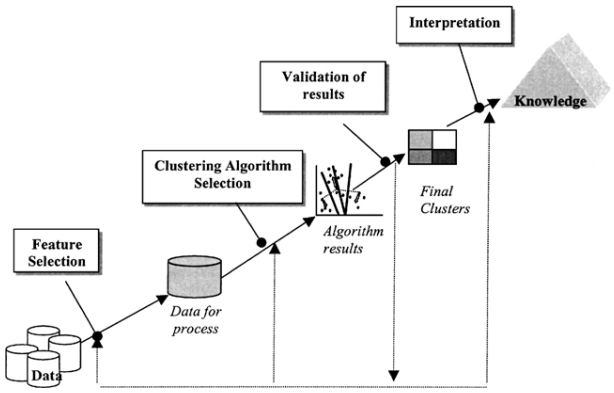

Figure 1: Steps of the clustering process [3].

All steps of the clustering process are depicted in figure 1. In this work we are not concerned with the clustering algorithm, but finding cluster in preprocessed data from a particle detector. The clusters to find are different events in high energy physics. The basic idea is, that certain objects produce a characteristic signal within a measuring apparatus. It is not yet of importance which features we are speaking of, but that these features cluster together.

It is our intention to develop an efficient and reliable method to find and separate these data clusters within any data set. We implemented three different methods with which we intend to find all data clusters.

While these methods aim to be as general as possible, they were specifically developed for use with Belle 2 data from the pixel detector. We will briefly introduce more general terms about Belle II in the coming subsections.

1.1 Belle II at SuperKEKB

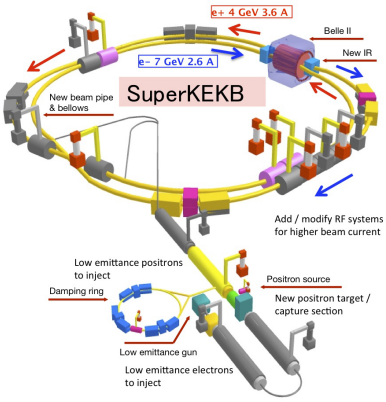

Figure 2: A schematic of the SuperKEKB collider facility [4].

The high energy physics research facility SuperKEKB, which is an upgrade to KEKB, is located in Tsukuba, about 60km outside of Tokyo. KEK is an abbreviation of the japanese name of the facility "Kō Enerugī Kasokuki Kenkyū Kikō". SuperKEKB is an Electron-Positron collider and Belle II is the second generation of a detector system built by an international collaboration. SuperKEKB and every one of its predecessors are so called B-factories. This simply means that KEK was built for the energy region in which B-mesons are produced.

Here is a list of all the sub-detectors [7]:

- pixelated silicon sensors (PXD)
- silicon strip sensors (SVD)
- central drift chamber (CDC)
- Time-Of-Propagation (TOP)
- another ring-imaging Cherenkov counters (ARICH)
- electromagnetic calorimeter (ECL)
- iron flux-return located outside of the coil (KLM)

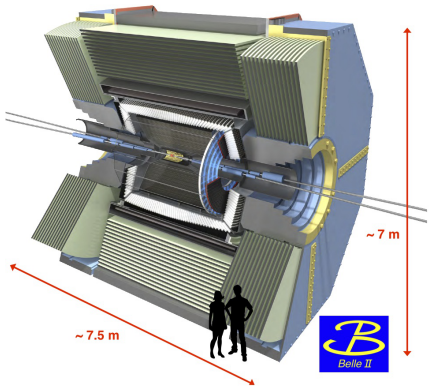

Figure 3: A 3D render of the Belle II detector with size indication and the Belle II logo [4].

Important systems for starting the data readout and storing are the Trigger and High Level Trigger system (TRG and HLT), the slow control and the data acquisition system (DAQ). Various sub-trigger systems (CDC, ECL, TOP, ARICH and KLM) send trigger information to a central trigger logic, often referred als Level 1 Trigger, where the decision is made whether to start data taking. After the Level 1 Trigger decides to start data taking, the DAQ starts over. The main purpose of the DAQ is to start the readout from the various subdetectors, processing and writing in the storage system. The data from every subsystem, except of the PXD, is sent to the Event Builder, where the data belonging together is merged into one event. The full reconstruction of the event including the PXD data is then performed by the HLT, which makes the final decision of keeping the event. The PXD module has to be treated separately due to his significantly higher data rate comparing to the other subdetectors, which is about 10 times higher than the data rate of all other sub-detectors combined. [8]

All of the information summarized in the following sections are taken from the master thesis by Katharina Dort [1], the PhD thesis by Thomas Geßler [9] and the techincal design report by the Belle II group [10].

1.1.1 The Pixel Sub-detector

The data processed for this work all are related to the PXD sub-detector. It consists of two layers of 40 silicon pixel sensors aranged concentrically and is not much larger than a soda can [5]. A schematic of it can be seen in figure 4. The PXD sub-detector is the closest detector to the beam. It is surrounded by four more layers of silicon stripes, which make up the silicon vertex detector (SVD).

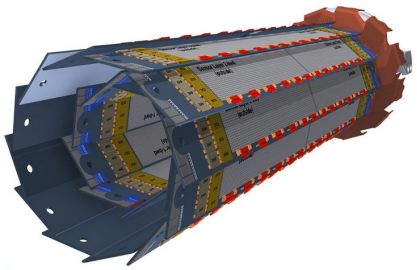

Figure 4: A 3D render of the PXD detector. One can see the two layer structure of it [5].

1.1.2 The data

In this work we used different kinds of data files. Firstly we created simple data files with clearly separated clusters. We started out with two dimensional data to develop the methods discussed in sections 3. Than we verified the approaches using the same sets, but with higher dimensions.

Since Belle II only just started taking data with the current detector setup, the data we used were simulated data files for the PXD. We have four clusters, labeled as PI, BG, T and DD and correspond respectively to pions, beam background, tetra quacks and anti-deuterons.

The files combined have 252.773 six dimensional points. Every entry of every point correspond to the number of pixels hit by a particle, the number of pixels in horizontal (u-direction) and in vertical (v-direction), the charge deposited, the minimum charge in an event and the maximal charge.

1.2 Software approach

Figure 5: The Python logo.

All methods of data analysis were developed using Python and the packages NumPy and SciPy. This gave us great flexibility since there was no need to compile and recompile the source code and NumPy and SciPy provided all

necessary functions and methods. All plots were generated by using Matplotlib and were printed directly after running the code.

2 Self-organizing map

2.1 What is a self-organizing map?

This is a form of unsupervised learning with the help of a neural network. The goal is to visualize high dimensional data in a two dimensional neuron grid and sort the input by similarity. In the end one should be able to associate each data point with one of the neurons and in turn know to which data cluster it belongs.

In the beginning of the process there are point clouds and usually one hundred randomly initialized vectors. Each of these vectors are associated with one neuron on ten by ten grid. A three dimensional illustration of this first step can be seen on the left side in figure 6. The choice of a ten by ten grid is rather arbitrary and some variations of self-organizing maps use different amounts of neurons, different shapes of grids (e.g. hexagonal) and even grow new neurons if needed.

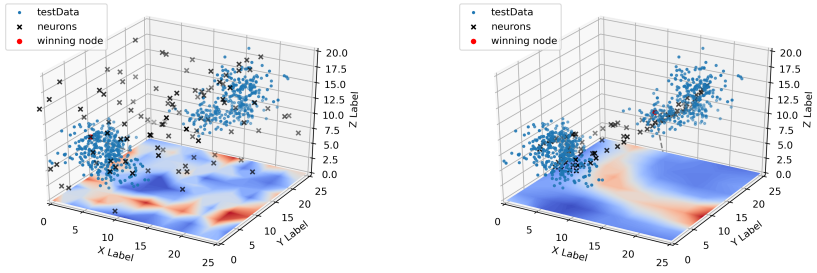

Figure 6: The first step of a self-organizing map. One can see the data clouds in blue, the vectors of each neuron as black x's and on the floor the U-matrix.

Every point of the data will be picked in random order and one calculates the closest neuron(often referred as winning neuron) and nudges this neuron a bit in the direction of the data point. But also every neighboring neuron is pushed ever so slightly in the direction of the data point. Over time the strength of the push decreases. The function of how much the nudge of the neuron in the direction of the input vector decreases is called the learning rate, which is an important property of the Self-organizing-map.

A common method to visualize the clusters in a self organizing map is the U-matrix, which stands for unified distance matrix. For each neuron the affiliated value in the U-matrix is the sum of the distance to its neighboring neurons divided by the number of the neighbors [11]. Before the training process of the map the U-matrix is in a random shape with no visible pattern. As each vector moves in the direction of one of the clouds, a pattern begins to emerge from the U-matrix. After looping through the data we expect the U-matrix to have areas with neurons very close to each other, separated by areas of neurons being very

far from each other. This can be seen in the x-y plane in figure 6, when one compares the first and last step of the map.

In the end one should end up with something as it is depicted on the right side in figure 6. One can clearly see that all but a few of the small x's have moved inside the clusters and that the rather random U-matrix from the illustration has turned into a more regular shape. There are now three distinct areas in the U-matrix, two with very low distance and one between them with a very high distance.

As mentioned before, not only the winning node, but also the neighbors of the winning node are modified to be more likely like the input vector. The function regulating how the neighbors are changed is called the neighborhood function. The most widely used neighborhood functions are the Bubble and the Gaussian function. While the Bubble function only changes the neighboring neurons around the winning neuron evenly, the Gaussian function decreases the value v with increasing distance to the winning neuron exponentially. In our implementation we tried a slightly different neighborhood function, defined in (1).

$$v_{ij}^{c} = \alpha(t) \cdot 2^{|i-i_c|+|j-j_c|} \tag{1}$$

Where $\alpha(t)$ is the learning rate in the timestep t, i and j the indices of the neuron and c the winning neuron. We also implemented the possibility of setting a cutoff radius, which determines the amount of neighbors that will be considered.

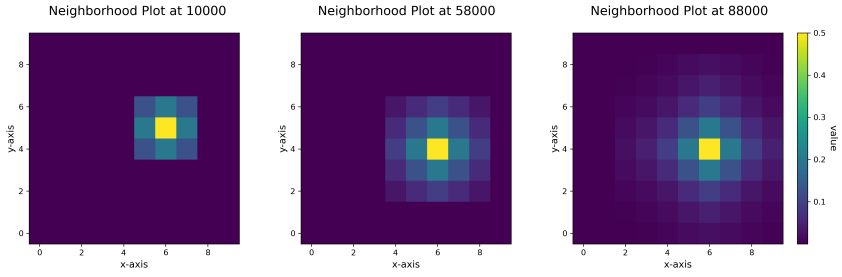

Figure 7: Neighborhood for cutoff 1 (l), 2 (m) and no cutoff (r)

In figures 7 and 8 one can see the difference between different cutoff radii. On the left in both figures we only change the direct neighbors of the winning node, than in the middle we also change the next neighbors and finally on the right one can see a smooth fall off, meaning that we take up to four neighbors into account.

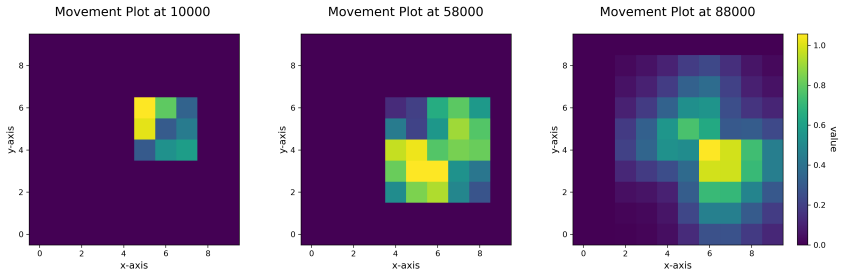

Figure 8: The corresponding movement plot to figure 7

2.2 Our results

First we generated different kinds of trail data against we verified our method. This consisted of two fairly wide separated point clouds and a series of different sized cubes. Our results, meaning the U-matrix can be seen below. For higher amounts of data it is advisable to use a higher amount of neurons.

2.2.1 Test Data

We generated different configurations of test data, which means varying data densities and numbers of clusters. In this section follows the results of these test runs.

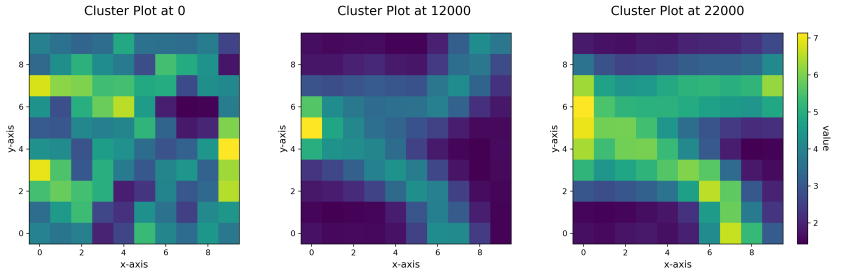

Figure 9: U-matrix for a six dimensional test data and 10×10 neurons at different steps.

In figures 9 and 10 one sees two tests with 10×10 and 15×15 respectively. Both figures show the first time step on the left, 12.000 in the middle and 22.000 on the right. We see in both cases that at time step 12.000 the U-matrix already shows the three clusters, but in figure 10 the clusters are more distinctly separated. But we see a known effect at time step 22.000 where a fourth cluster is beginning to develop. This effect is know as overtraining and it is visible in figure 10 [12].

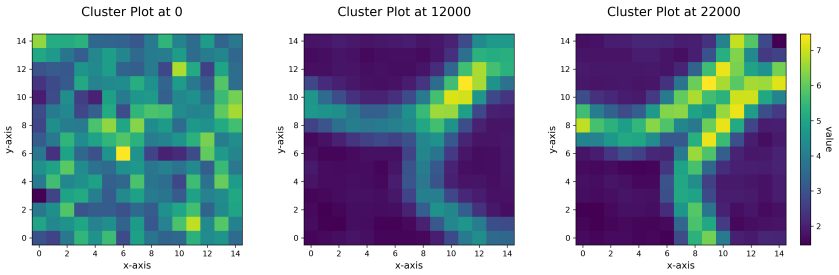

Figure 10: U-matrix for a six dimensional test data and 15×15 neurons at different steps.

2.2.2 Simulated Data

Further we tested different sets of the simulated PXD data. Theses simulated data included pions (PI), anti-deutorons (DD), tetraquarks (T) and beam background (BG). We run tests with all combinations of each simulated data set.

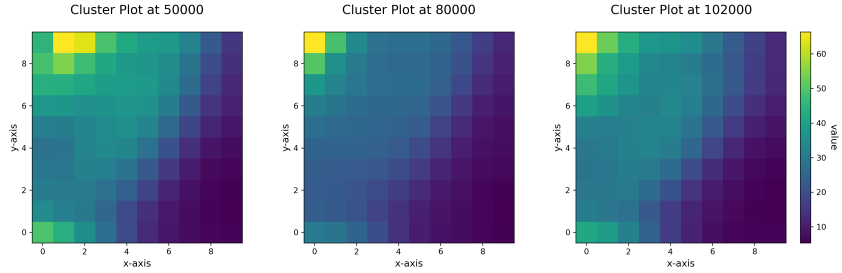

Figure 11: The 15×15 U-matrix at time steps 50.000, 80.000 and 102.000 for PI and DD.

Figure 11 shows the results for PI and DD. This data set consists of 103.429 points. In the plot on the left and right we clearly see a single cluster in the lower right, stretching to the top right. Slight signs of a second cluster can be seen on the left side in the lower half of the y-axis between 2 and 4. We would need to adjust the learning rate or simply process more data point, as it seems it did not finish its learning process.

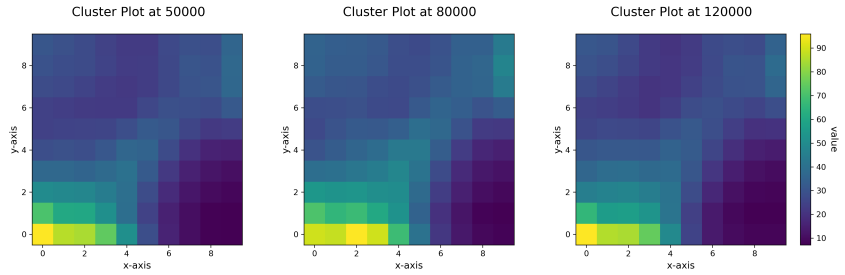

Figure 12: The 15×15 U-matrix at time steps 50.000, 80.000 and 120.000 for PI and T.

Figure 12 shows the results for PI and T. This data set is larger compared to the one for PI and DD. We again see a clear cluster in the lower right corner and we can make out a second cluster in the upper left corner. Maybe adjusting the learning rate would improve the quality of the second cluster. This set has 120.671 data points and thus has 17.242 more data points.

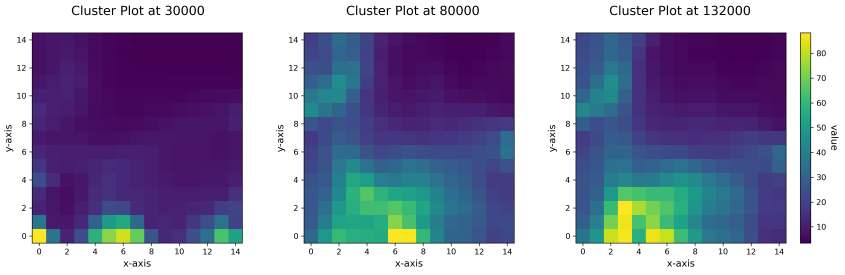

Figure 13: The 15×15 U-matrix at time steps 30.000, 80.000 and 132.000 for DD and beam background.

Figure 13 shows the results for DD and beam background data. The first frame looks like it is just one large cluster with barely any structure. The plot in the middle and on the right show a cluster in the upper right corner of the U-matrix and one can make out three more clusters, even though we would expect only two clusters. This data set consists of 132.102 data points and since we see four clusters already at time step 80.000, we assume the network is not overlearnt, but that we still did not have enough data points or have to tweak the learningrate or the neighborhood function.

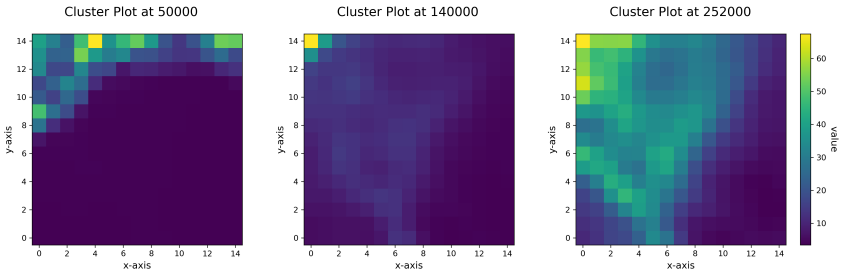

Figure 14: The 15×15 U-matrix at time steps 50.000, 140.000 and 252.000 for all data sets.

Figure 14 shows the results if one uses the self organizing map on all data sets. The first plot in the left shows one large cluster in the U-matrix, which slowly shows some structure in the second plot in the middle. This structure becomes more pronounced in the last frame on the right. We can see two clusters, one right side and the second in the lower left. Within the lighter structure one might be able to make out the beginnings of three more clusters. Since we would expect four clusters and we have sufficient many data points, we are assuming, that we not only would need to adjust the learning rate, but also increase the number of neurons.

2.3 Conclusion

As it was said above, this method works reliably with clearly separated clusters, as it was the case with our simple test data sets. But as soon as we used the simulated training data sets, this Self-organizing map started to fail the data separation. A simple reason for that could be the order in which the data is processed. The simulated data is not as good separated as our test data sets, with the clusters having common points and possibly even merging into each other in some dimensions. There are settings as the neighborhood function, learningrate and the amount of neurons that significantly influence the functionality of the Self-organizing-map. One could make many test runs using various combinations of these setting to maybe yield better results.

3 High dimensional voxels

3.1 What is a voxel?

It is the easiest to understand what a voxel is, when one thinks of pixels. A pixel is a color dot, which is sitting on a regular grid. It is a shortening of 'picture element'. One can understand a voxel in the very same manner, the difference being, that a pixel is two dimensional, while a voxel is three dimensional. Voxel is short for 'volume element'. In this work we use it a bit more loose, we not only mean three dimensional volumes, but up to six dimensions.

3.2 A stochastic approach

We tried a stochastic approach, where we picked random data points and grew spherical voxels. The first problem we encountered was the question of how large a voxel should be and until what point should we let it grow. This turned into the question of how one should define the boundary and how to find the boundary. Furthermore spheres do leave gaps in between them, no matter how small or close they get to each other. How do we finally define what a cluster is? These three points will be discussed in this section.

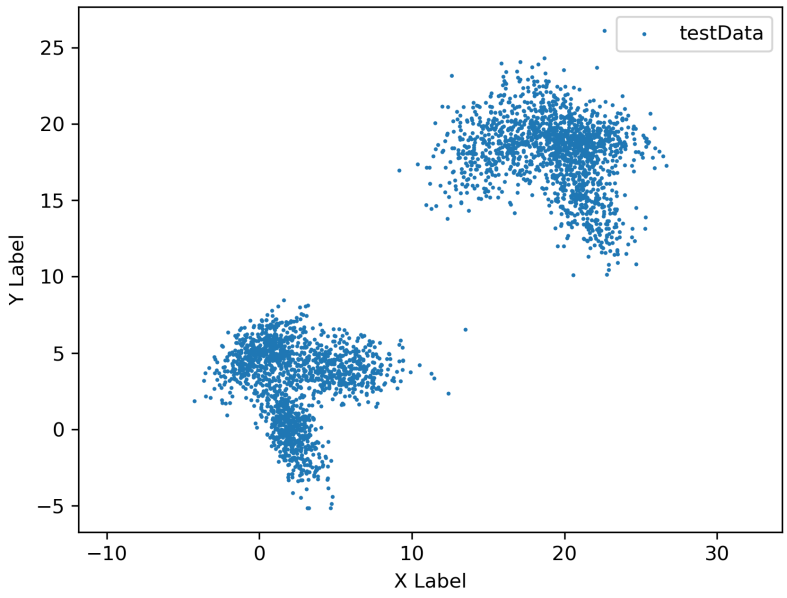

Figure 15: Unprocessed two dimensional data.

This method was developed using a two dimensional test data set, of which a

plot can be seen in figure 15. This data consists of two point clouds with varying density and an irregular shape.

3.2.1 Finding the boundary

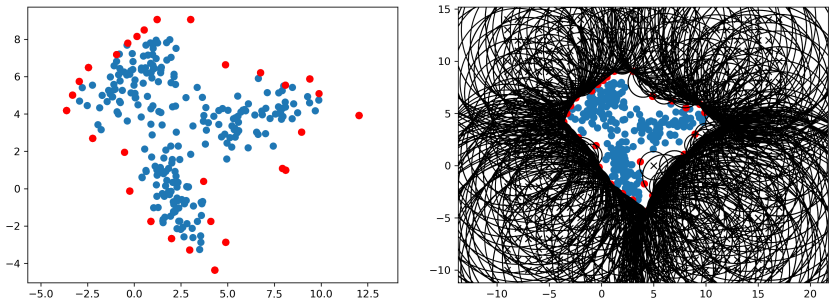

Figure 16: Illustration of finding the boundary of data clouds. On the left: a cloud with boundary points in red. On the right: the randomly grown voxels for finding the boundary.

In figure 16 one can see an illustration on how the program tries to find the boundary. We randomly place voxels in the whole space of the data set, than we check within a certain radius, which is arbitrarily set by the user, if there are any data points in the vicinity of that voxel. We call this radius the search radius. In the case of there begin no data points, the program searches the nearest neighbor and flags it as a boundary point. How many times the algorithm does is also set by the user. The number of checks are called trails.

3.2.2 Growing Voxels

We want to illustrate how this algorithm works, for that one should have a look at figures 16 and 17 and follow the steps along. First the algorithm is using a stochastic approach at finding the boundary.

Then the second step after the boundary was found, the algorithm loops through all data points looking for the largest voxel possible. It does this by checking against every boundary point and every other voxel. Now if the distance to the next voxel is smaller, than the preset search radius for finding the boundary, this newly set voxel will grow until it hits its neighboring voxel. In figure 17 the voxels with numbers $0, 1, 2, 3$ and 4 explain this process well. Voxel 0 stops growing at a boundary point, voxel 1 as well, since it grows inside another cluster. Subsequently all following voxels align themselves with the preceding voxels. This step is finished until a newly created voxel only contains a single point.

In the third step the algorithm picks random points from the data set and places new voxels, growing and moving them until they connect two voxels with each other.

The fourth and last step is finding the clusters and it will be explained in the next subsection.

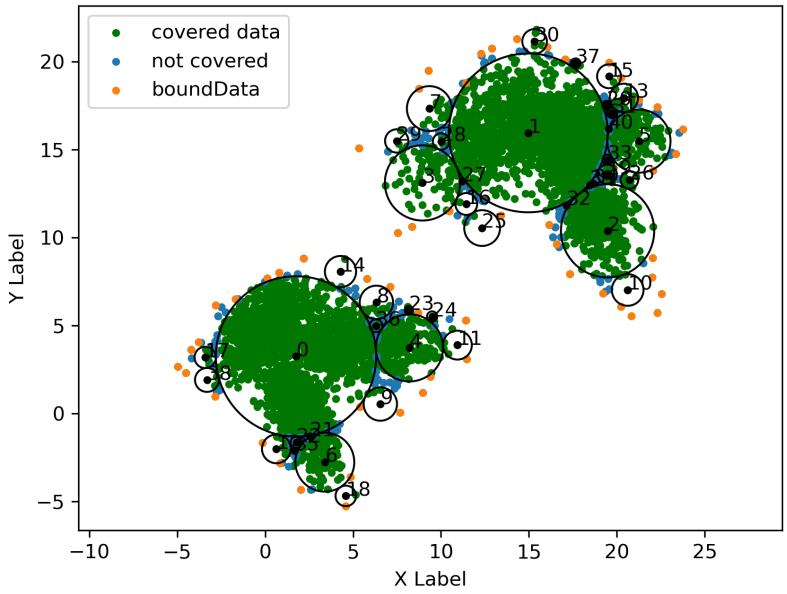

Figure 17: Process of placing the largest possible voxels on two dimensional data clouds.

The full results of the test run in figure 17 is as follows. It consists of 41 voxels, the largest voxel has a radius of 4.56 units, while the smallest has a radius of 0.04 units. The voxel with most counts has 2074 and the lowest count is 2. Furthermore the algorithm covers about 5000 data points, which translates into 95.46% of the data. We found two clusters, with the first consisting 17 voxels and covering 48% and the second 24 voxels and covering 47.5%.

3.2.3 Finding the voxels that belong together

In order to find which voxels build up a cluster, one has to find which voxels are connected together by chains of neighbors. In figure 18 one can see two data clouds and a few voxels, each with their respective voxel number.

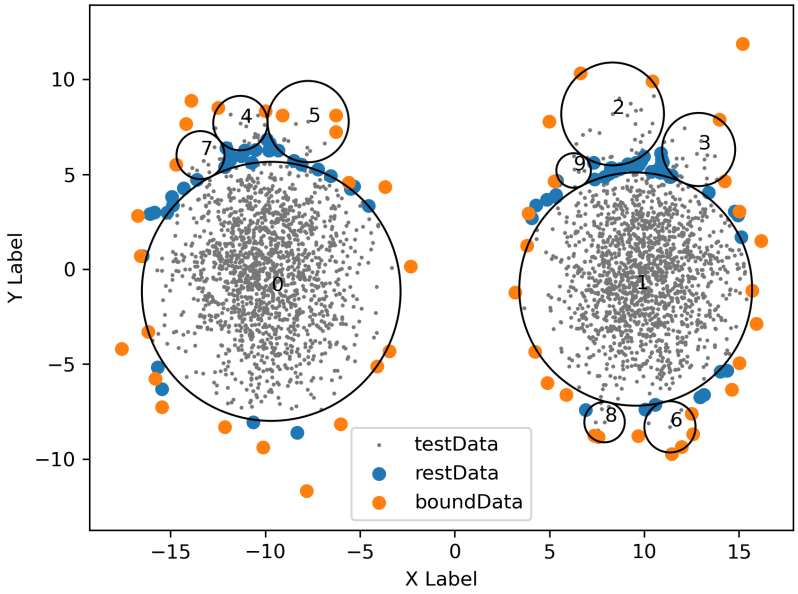

Figure 18: Explanation of the method to finding clusters with the help of voxels.

This method now writes a list for each voxel, containing itself and all other voxels it touches. Than it compares these lists and combines them, if at least one element is in common. The third and last step throws all non-unique lists away. The number of resulting lists give us the number of data clusters, given that the clusters are sufficiently far apart in at least one dimension. For the example in figure 18 one gets the following:

$$\left.\begin{array}{l}[0,7]\\ [1,3,6,8,9]\\ [2,9]\\ [3,1]\\ [4,5,7]\\ [5,4]\\ [6,1]\\ [7,0,4]\\ [8,1]\\ [9,1,2]\end{array}\right\} \Longrightarrow \left.\begin{array}{l}[0,4,5,7]\\ [1,2,3,6,8,9]\\ [1,2,3,6,8,9]\\ [1,2,3,6,8,9]\\ [0,4,5,7]\\ [0,4,5,7]\\ [1,2,3,6,8,9]\\ [0,4,5,7]\\ [1,2,3,6,8,9]\\ [1,2,3,6,8,9]\end{array}\right\} \Longrightarrow \begin{array}{l}[0,4,5,7]\\ [1,2,3,6,8,9]\end{array}$$

3.2.4 Testing on different test data sets

In order to get a better sense of this method and to find its weaknesses, we tested it on different kinds of data. We varied the sizes, densities and distances between the data clusters. In figures 19 and 20 are the plots for different test data sets.

For the left and right plots in figure 19 we used the same data set, the only difference was, that we changed the parameters for finding the boundary, meaning we changed the search radius but kept the number of trails constant. This resulted in vastly different interpretation of the data by the program.

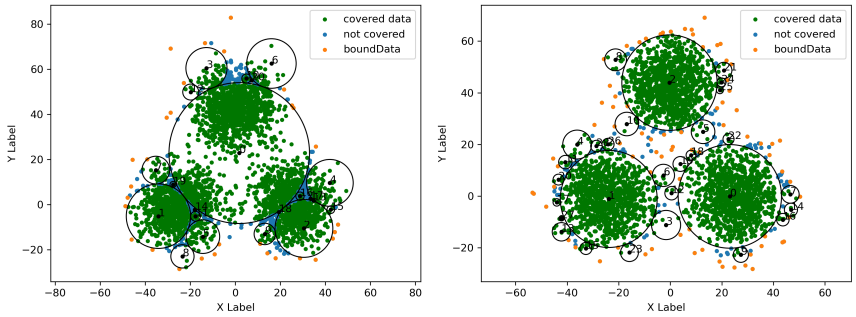

Figure 19: Demonstrating what impact the input parameters have on finding the boundary and thus on the end result.

The results are summarized in table 1. On the left side the largest voxel is nearly fifty percent larger than the largest voxel on the right side, it contains nearly sixty percent more data. The dominance of the one voxel on the left side makes it impossible to differentiate between the three clusters, since the first voxel sits firmly in the middle of all data points. The program can only barely recognize the three clusters on the right side, after we lowered the search radius from 7.75 down to 2.5.

Table 1: Comparing the results for different parameters in finding the boundary.

	Search-radius	Number of voxels	Data covered	Number of clusters	Largest radius	Highest count
Left	7.75	21	92.38%	1	31.21	1630
Right	2.5	29	94.63%	3	20.04	992

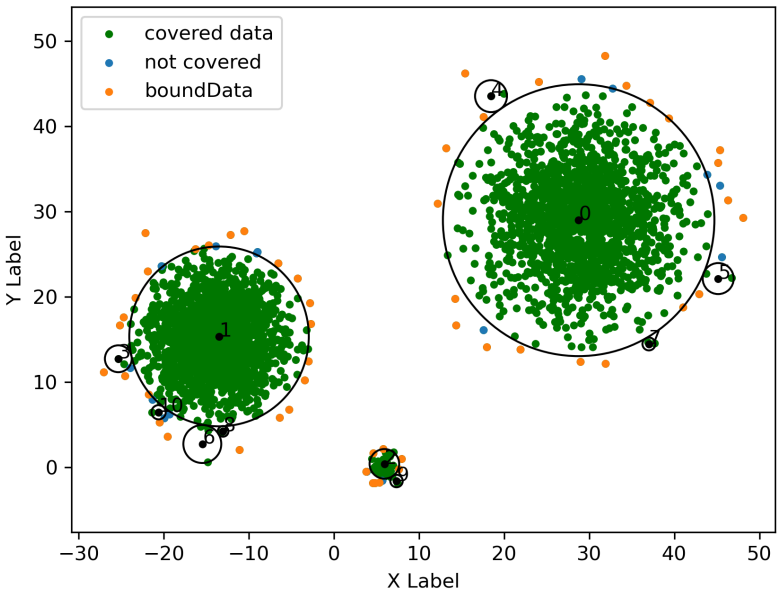

Figure 20: Testing the methde of different sized clouds, with differing densities.

The second data set we tested can be seen in figure 20 and the results are listed in 2. It is a collection of three different clusters of different sizes and with different densities. The whole data set consists of three clusters and 3355 data points and found 52 boundary points. There are 11 voxels in total and 97.97% of all data points where covered. The algorithm determined that there are three clusters. In all three clusters, the largest voxels had the highest density and these voxels were created back to back. Meaning before any voxels could touch each other, all clusters were covered by one voxel each.

Table 2: The results of the algorithm on three different clusters.

	Number of voxels	Highest density	Largest radius	Smallest radius	Highest count	Lowest count
Cluster 1	4	101.37	15.95	0.76	1617	2
Cluster 2	5	152.54	10.55	0.63	1609	2
Cluster 3	2	24.74	1.74	0.73	43	2

3.2.5 Results for higher dimensions of test data

We generated data sets with two clearly separated clusters of almost circular asymmetric shape at different dimensions. We kept the number of data points constant at 5240 and we kept the shape of the clusters similar.

The amount of data that was covered by this method decreased with an increase of dimensions. This was to be expected since the spaces in between the spheres increases in size with every dimension. This effect is related to the curse of dimensionality [13].

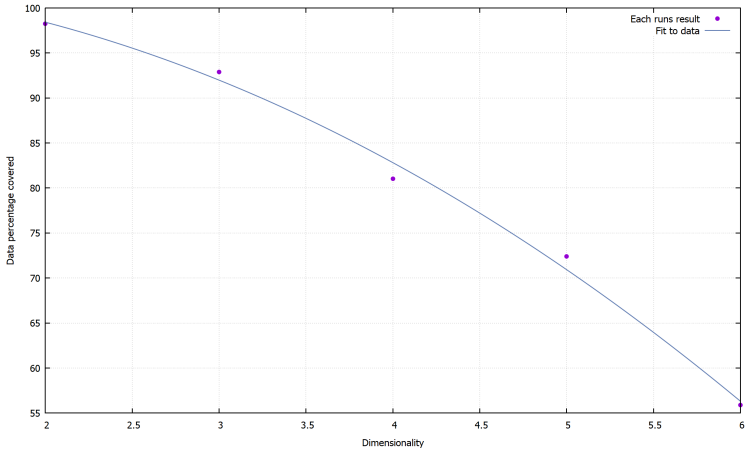

Figure 21: Data loss as a function of dimensionality.

Figure 21 shows how much data gets lost with an increasing dimensionality of the problem. To get a better understanding of the loss, we fitted it using a quadratic function and we can conclude that the loss is a quadratic function of the dimension. The data is tabulated below 3.

Table 3: Results for different dimensions.

Dimensions	Voxels	Clusters	Data Covered	Data Points
2	29	2	98.26%	5149
3	36	2	92.90%	4868
4	76	2	81.03%	4246
5	25	2	72.39%	3793
6	41	2	55.88%	2928

As we can see in table 3 the method may have lost coverage, but it still could figure out how many clusters there were. One curiosity is, that the number of

voxels stays relatively constant, except in the four dimensional space. In this case we have twice to thrice as many voxels as in the other cases. This might be connected to the fact that $\mathbb{R}^4$ is exotic.

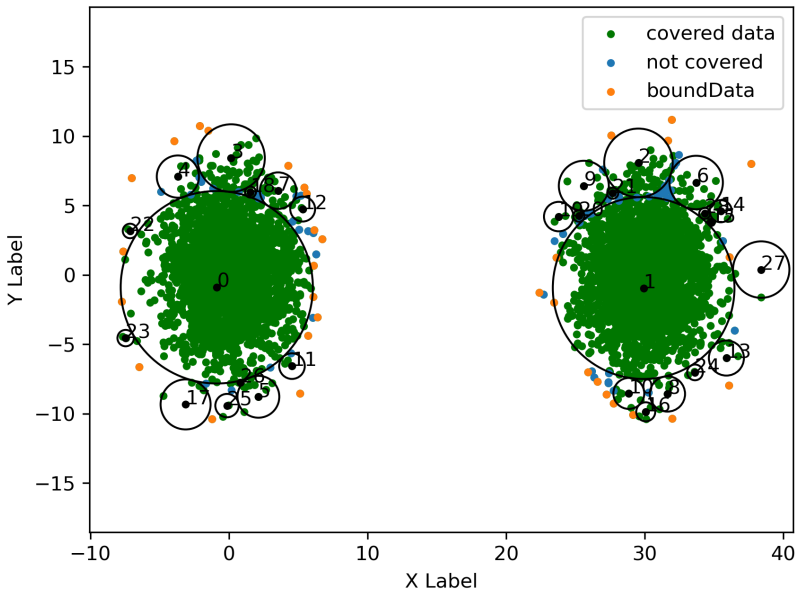

Figure 22: The two dimensional data set, that was analyzed.

In figure 22 we plotted the two dimensional data sets. Every higher dimension was created in the same way, in order to know approximately what the shape of theses clusters are.

The general findings of our algorithm can be found in table 3. Since the bar charts for the two through six dimensions look similar, we will only look at the second and sixth dimension.

It is very instructive to look at different statistics and visualize them. This is especially necessary when one tries to analyze unknown data of higher dimensions, since one cannot plot the data directly.

Figure 23 shows bar charts of the process. On the x-axis one sees the number of the voxel and on the y-axis one sees different characteristics of the voxel. In orange are the results after the first step of creating the largest possible voxels and in blue we have the results after randomly creating voxels.

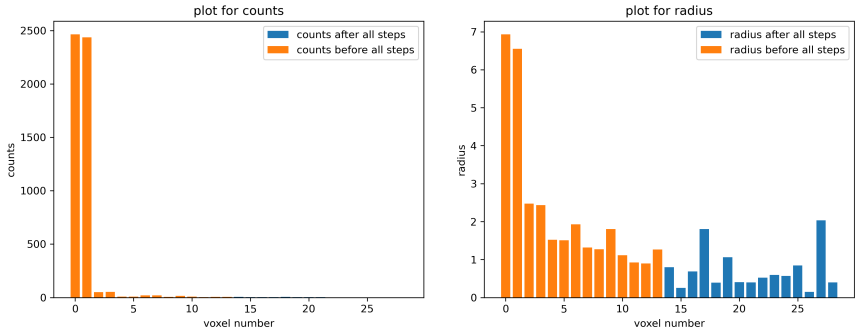

Figure 23: The statistics bar plots for the analysis of the two dimensional data set. On the left we can see the counts of data points per voxel and on the right we see the radii.

In figure 23 we see on the left, that two voxels contain almost all of the data points, meaning the first two voxels dominate. Looking than at the radii on the right, we see something similar, the first two voxel are the largest voxels and than the size tapers off until the second step, where we start growing stochasticly placed voxels.

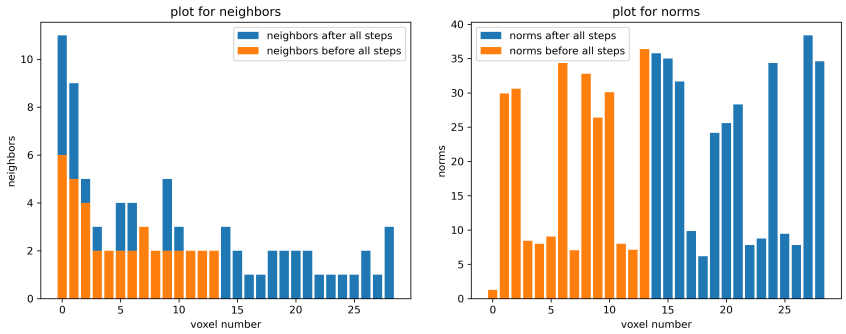

Figure 24: The statistics bar plots for the analysis of the two dimensional data set. On the left we see the number of neighbors per voxel and on the right we the the norms of the midpoints.

Figure 24 again gives some insights. On the left we see the number of neighbors each voxel has and as expected the number of neighbors will increase for every voxel created in the first step. The more interesting plot can be seen on the right side. Here we plotted the norms of every midpoint of all voxels. Norms meaning here the euclidean distance from the origin of the coordinate system. This plot clearly reveals that there are two data clouds and that they are quit far apart, meaning 30 units. This is also what the algorithm found,

the distance between the weighted midpoints of each cluster is, according to the algorithm, 30.76 units.

We can correlate that finding with the plot in figure 22, where we can see, that the clouds are centered around zero and thirty on the x-axis and have a width of about ten units, meaning the stretch five in each directions. The right plot in figure 24 reveals that the two largest voxels have radii between six and seven units.

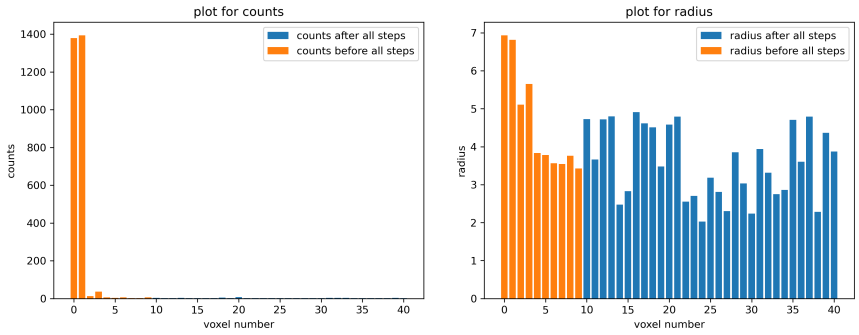

Figure 25: The statistics bar plots for the analysis of the six dimensional data set. On the left we can see the counts of data points per voxel and on the right we see the radii.

Now for the six dimensional case it looks akin to what we have in all other cases. In figure 25 we again see, that two voxels dominate. The biggest difference here is, that the radii are more evenly distributed.

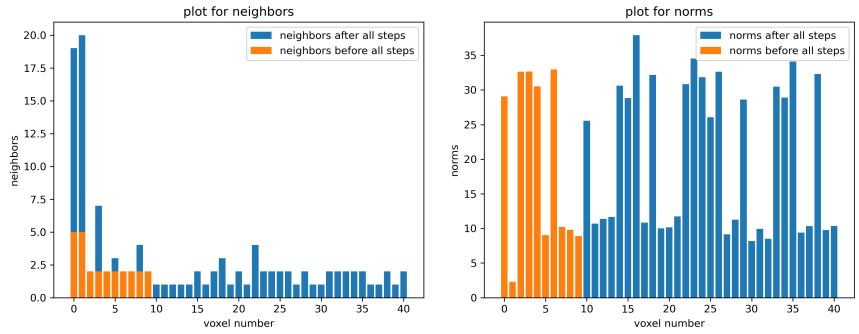

Figure 26: The statistics bar plots for the analysis of the six dimensional data set. On the left we see the number of neighbors per voxel and on the right we the the norms of the midpoints.

As for the numbers of neighboring voxels and the norms of the midpoints, figure 26 paints a nearly identical picture, the difference begin, that the neighbors

for the first two voxels increased four fold and not two fold.

These results were to be expected, because the data clouds for all test data were of the same shape. The increase in neighbors also makes sense, since we have twelve more voxels and with higher dimensions the surface of each voxels increases, meaning we have more possibilities for more neighbors. The more evenly distributed radii in the six dimensional case is a bit off.

3.2.6 Limits in distance differentiation

We also tested what the limits of this method are, by using the same data set, but moving the clouds closer together. We started with the midpoints of the clouds are twenty units apart in the beginning and each cloud having a mean diameter of 10 units. Then moved them closer together, without changing the diameter. In figure 27 are the results for 17 and 16 units in the two dimensional space.

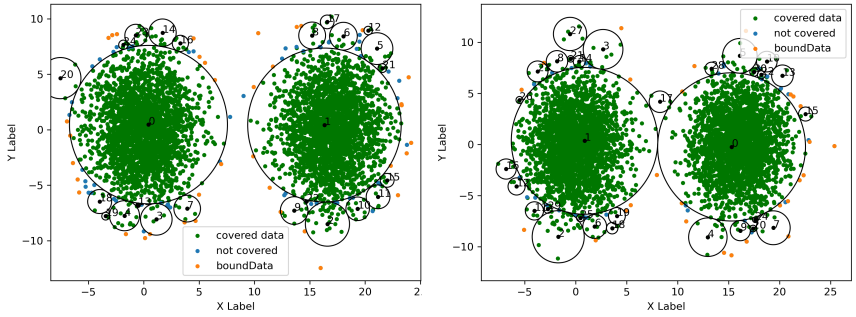

Figure 27: Varying the distance between clusters in two dimension.

What we see is, that at 17 units we can still recognize the two clouds, but at 16 this method fails. We also observed, that in higher dimensions, the clusters needed to be further apart in order to tell them apart with this method. In a three and four dimensional space we lost the ability to differentiate at 16 units. In a five dimensional space we needed 17 units, while a six dimensional space required 24 units to tell them apart.

If we evaluate the norms of the midpoints, we can conclude, that there are different data clouds. This approach of considering the norms only works properly, if the norms are evenly distributed. For this we just have a look at the norms bar chart for six dimensions, which can be seen on the left in figure 28. As stated above, the clustering algorithm of looking at neighbors fails, while we clearly see two clusters, when we just take the norms into account. One possible conclusion is, that the main voxels have grown too large. But we cannot say that conclusively if we correlate this with the radiuses for voxel zero and one, which can be seen on the right side in figure 28.

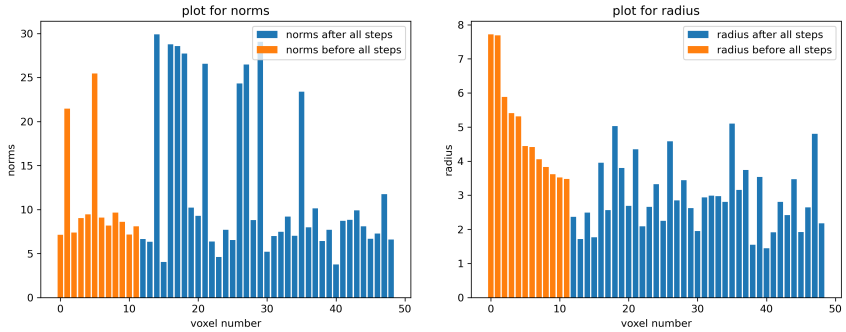

Figure 28: The statistics bar plots for the analysis of the six dimensional data set. On the left we see the norms of the midpoints of each voxel and on the right we see the radii per voxel.

3.2.7 Analyzing the simulated PXD data

In this section we go over the results for the analysis of the simulated PXD data. We have four sets, pions (PI), anti-deuterons (DD), tetraquarks (T) and beam background (BG). The PI set consists of 74,756 data points, the DD of 28,673, the T of 45,915 and the BG file of 103,429.

A brief summary of the results of all combinations can be seen in table 4. We analyzed the the combinations PI with DD, PI with BG and PI with T.

Table 4: The results of the spherical voxel method for different combinations of the simulated PXD data.

	Data points	Border-points	Voxels step 1	Coverage step 1	Voxels step 2	Coverage step 2
PI+DD	103,429	600	19	46.99%	77	71.28%
PI+BG	178,185	673	11	5.50%	45	93.72%
PI+T	120,671	678	18	33.17%	73	62.79%
DD+T	74,588	1042	31	62.97%	125	72.30%
DD+BG	132,102	1785	6	2.95%	25	92.38%
T+BG	149,344	1531	20	15.46%	81	83.68%

The algorithm found for combinations but the beam background ones only a single cluster. The second cluster found in theses cases consisted only of single digits numbers of voxels and contained less than one percent of data data. This is why we will just neglect them.

Example: (Pions and anti-deuterons)
Now we will review the three cases where PI are combined with the other cluster files. The results of the other variations follow a very similar structure, which is why it is not too instructive to look at every single case in detail. The first example is PI+DD. This combination contains 103,429 data points in total.

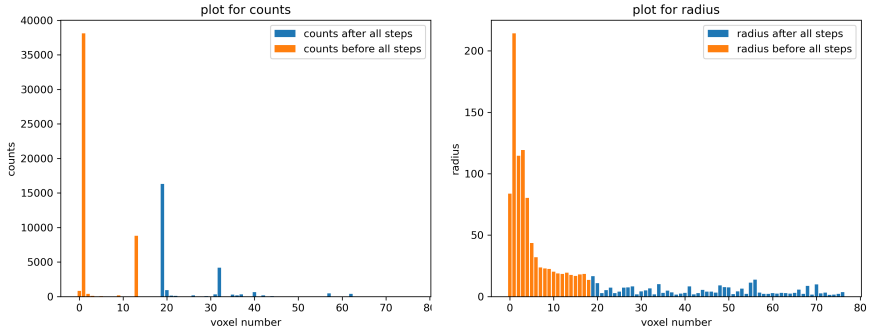

Figure 29: The statistics bar plots for the analysis of the simulated data sets for PI+DD. On the left we can see the counts of data points per voxel and on the right we see the radii.

The algorithm created 19 voxels in the first step and covered 46.99% of the data. In the second step it created 58 more voxels totalling at 77 and covering 71.28% of the data. The largest radius is 214 units and the mean radius is at 15 units. The highest count for data points is 38,121 with an average of 958. The clustering algorithm recognized the whole set as only one cluster.

In figures 29 and 30 we see the results for the Pion anti-deuteron data set. Again the radii decrease, after an initial peak. This effect is due to the the fact, that from the second voxel onward, voxels can grow beyond the boundary as long as the growth rate states below a certain threshold. The second voxel is by far the largest and also contains the most data points. Which could indicate, that it might be covering one cluster almost by itself. Since we allow voxels to grow beyond the border under certain conditions, we can conclude, that the second voxel overgrew, touching the first voxel.

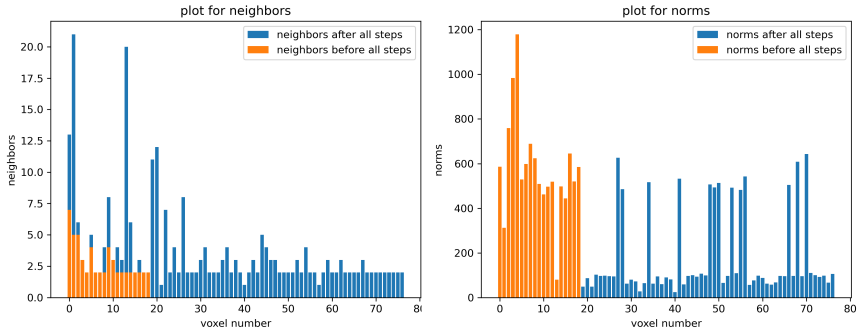

Figure 30: The statistics bar plots for the analysis of the simulated data sets for PI+DD. On the left we see the number of neighbors per voxel and on the right we see the norms of the midpoints.

Figure 30 shows some interesting facts about the process. Firstly we see, that the largest voxels also have the most neighbors. It is a indication, that thises voxels are central or at least a lot of voxels were created around them.

The norms, meaning the euclidean distance to the origin of the coordinate system, of the midpoints show something interesting. Even though the algorithm found only one cluster, meaning all voxel could be connected via a chain of voxels, the norm plot in figure 30 has very distinct values. Meaning one would be able to find two clusters and tell them clearly apart. The second voxel sits right in the middle of the two values for all other voxels. So we are assuming, that this voxel is connecting the two clouds.

Example: (Pions and beam background)
For the PI+BG analysis we found two clusters, yet one of them had less than one percent of all the data. This was a characteristic that all beam background analysis yielded. The first step created 11 voxels and covered only 5.5% of the data. After the second step 45 voxels were created in total, covering 93.77%. In the end we had only four times as many voxels than in the first step, but 18 times as many counts. This was repeated for the DD+BG and T+BG combinations, as can be seen in table 4.

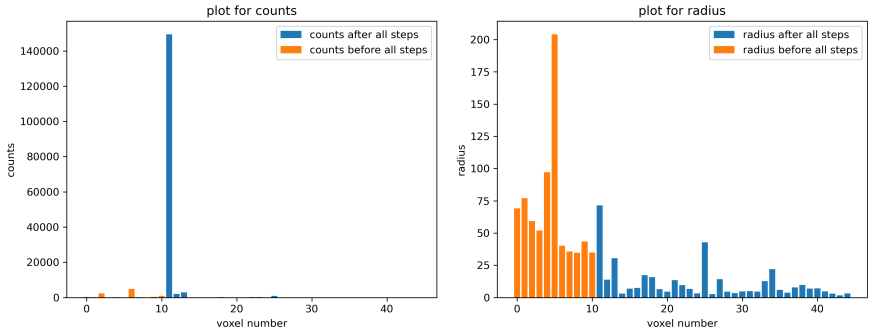

Figure 31: The statistics bar plots for the analysis of the simulated data sets for PI+BG. On the left we can see the counts of data points per voxel and on the right we see the radii.

Figure 31 shows a characteristic result for beam background analysis. In all cases, where we combined beam background with another cluster, the first step created few voxels, covering only a low amount of data points. The second step creates a lot of voxels, covering way more of the data. All of them had a peak in counts on the first randomly created voxel, covering most of the data.

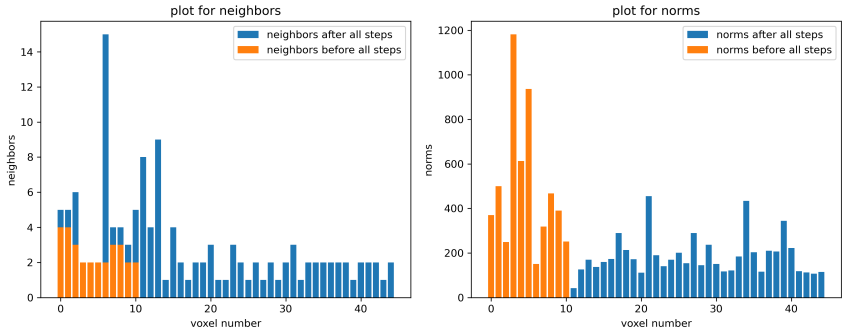

Figure 32: The statistics bar plots for the analysis of the simulated data sets for PI+BG. On the left we see the number of neighbors per voxel and on the right we see the norms of the midpoints.

Figure 32 shows the numbers of neighbors and the norms. Neither of them reveal anything deeper about the process, since the number of neighbors is mostly between six and one, dominated by just two neighbors. There are peaks at eight, nine and fifteen neighbors. The norms do not show any pattern, unlike what we have seen in the previous analysis.

Example: (Pions and tetra quarks)

Finally we will have a more detailed look at PI+T. This set has 120,671 data points for which 678 boundary points were found. After the first step 18 voxels were created and 33.17% of the whole set was covered with voxels. The second step created 55 voxels totalling in 73 voxels and covering 62.79% of the data. As we can see in table 4 the two data sets were recognized just one cluster.

More detailed information from the program output tells us that the highest count is 29,088 and the average count is 1037.88. The largest radius is 118 and the mean radius is 19.5 units.

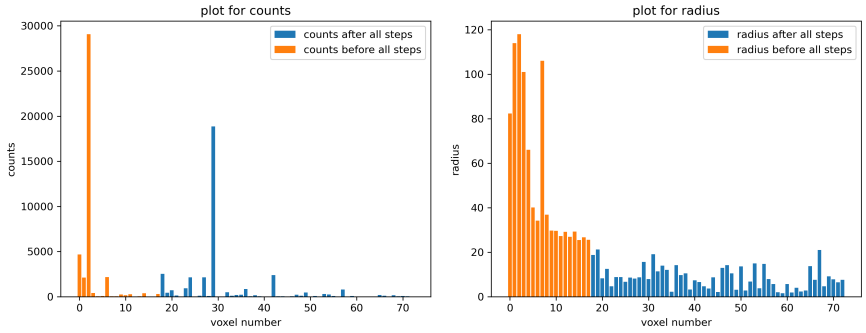

Figure 33: The statistics bar plots for the analysis of the simulated data sets for PI+T. On the left we can see the counts of data points per voxel and on the right we see the radii.

Figure 33 shows the counts on the left and we can conclude that one of the first-step voxels and one of the second step voxels contain most of the covered data. This is not an indication that we have just two cluster, it would have been, if two of the first-step voxels had covered most of the data. On the right we see the radii, which start out large and temper down and the second step voxels are pretty small in comparison.

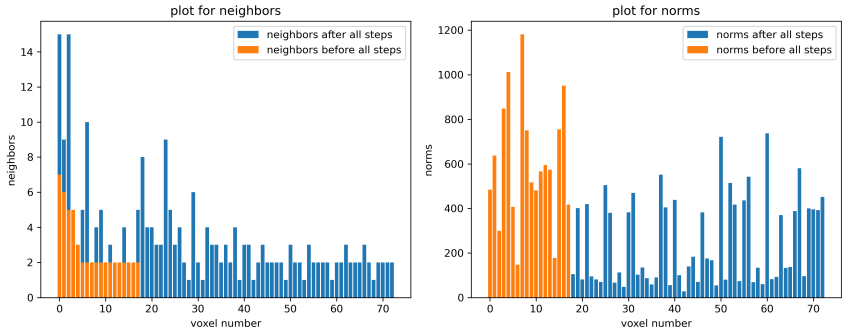

Figure 34: The statistics bar plots for the analysis of the simulated data sets for PI+T. On the left we see the number of neighbors per voxel and on the right we see the norms of the midpoints.

Figure 34 shows the number of neighbors on the left and the norms of the midpoints of each voxel. One interesting observation is, that some of the randomly grown voxels from step two have quit a lot more neighbors, than the first-step voxels.

In the norm plot we see the same thing we already saw in with PI+DD. There are two centers. We can only reiterate that this is a pretty good indication that there are two clusters and not just one and that the cluster algorithm failed.

3.3 A systematical approach

With the problems mentioned in the above paragraph we tried a systematical approach and used cubical voxels, which we arranged on a regular grid. The main benefit of cubical over spherical voxels is that there are no interspaces and overlaps. In case we have a grid that covers the whole data cloud, we can assume that all of the data is inside any voxel. The easiest and most effective way would be to discretize the space and place a voxel in evenly defined distances between the maximum and minimum of the coordinates in each dimension. Afterwards check if the placed voxels contain any data and delete them if not. Since we are working with multidimensional data in a wide range, a discretization over the whole data range in every dimension is not realizable due to the big computing effort. By increasing the dimension the discretization gets combinatorial explosive. So we need a method to find a grid that only covers the data points without the need of checking the empty spaces around the data clouds. Therefore, after choosing a step size, we loop through every coordinate in every dimension of of the data points and use the position that is closest to a multiple of the step size from the coordinate. In that way we get a grid that is approximately only covering the data cloud. Like before, for visualization and testing the method we used the two-dimensional test data set. The figure 35 shows the two-dimensional data plotted with red dots as the midpoints of the squares created in the next step. For comparsion figure 36 shows the midpoints of the potential voxels with the discretization of the whole space.

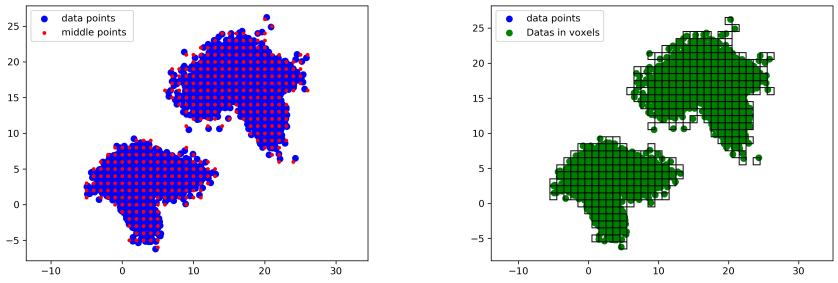

Figure 35: Midpoints and the squares created in the next step by looping over the data.

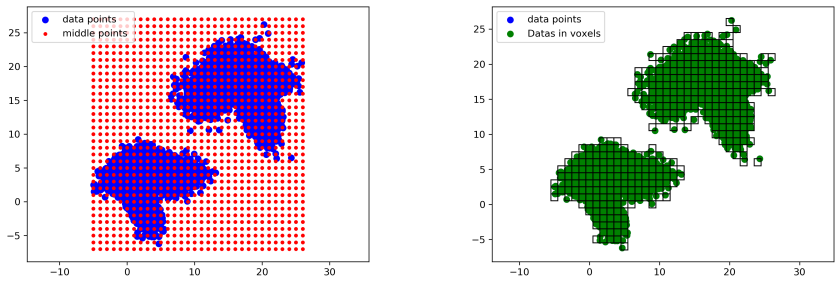

Figure 36: Midpoints and the squares created in the next step by discretization of the space.

Since we are rounding the coordinates to the next multiple of the step size to find the midpoints of the squares, it is possible to miss data points, because the square is created shifted to them. Various testing with this methods including higher dimensional data sets showed that most of the data is covered by the voxels using this method. In the two-dimensional case shown in figure 35 100 % of the data was inside any voxel.

3.3.1 Detecting the voxels that form a cluster

For finding the the voxels that form a cluster we use the same algorithm described in chapter 3.2.3. In this case we considered cubes as neighbors if their midpoints are half of the step size apart from each other. As it is observable in the figures 35 and 36, it is possible that detached voxels only containing one data point without having any neighbor are created. This occurs more often the smaller the step size is. These can be ignored and considered as loss of data for this method, as we cannot assign them to a bigger cluster. In the following we only take clusters consisting of at least two voxels or containing more than 0.1% of the complete data into account.

3.3.2 Choosing the step size

The main problem with this method is to find a suitable step size, which is equivalent to the edge of the cube. It is not only a matter of the computing effort, but also crtitical to the success of this method. Choosing a value for the step size that is too big or small leads to data loss as previously described. On the other hand it is not possible to separate the clusters if the step size is too big for the voxels from different clusters not to build neighborhood relationships and therefore separate them with the method introduced in chapter 3.2.3. To demonstrate this issue, we created a two-dimensional test data set with three clusters and ran the program with three different step sizes for the same data. Figure 37 shows an example for a step size that is compatible for this data

set. The squares are big enough to cover the whole data and not that big for voxels from different clusters touching each other. In this case three clusters were identified.

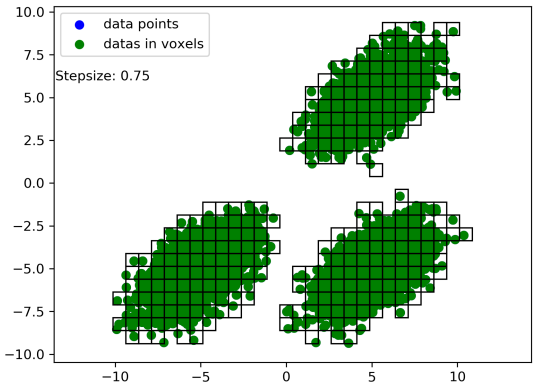

Figure 37: Example for separating three clusters with a suitable step size.

The step size chosen in figure 38 is just small enough for the algorithm to separate the cluster and can be viewed as a borderline case. Although the corners from voxels out of the two separate lower clusters are touching, they are not considered as neighbors, which is why the algorithm still detects three different clusters.

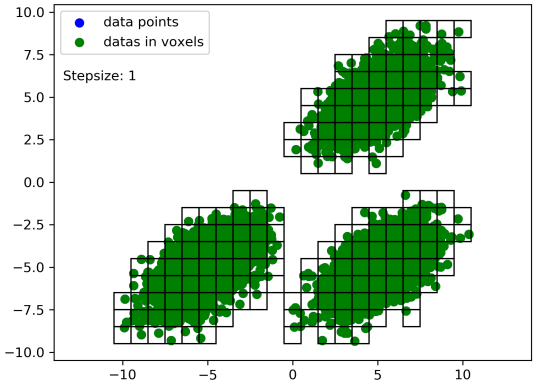

Figure 38: Borderline case with a step size just small enough to detect three clusters

By going further and increasing the step size even more, we observe the voxels from the lower clusters growing into each other, thus the algorithm can only detect two clusters. This illustration is shown in figure 39.

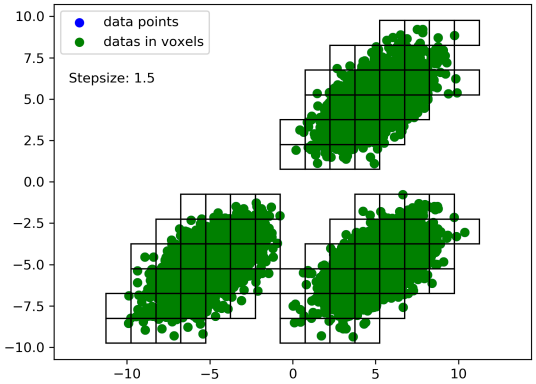

Figure 39: Example for a step size too big for detecting the three clusters

Approaches for finding an adequate step size for any data set, like considering the middle distance between neighboring data points as guideline for the edge length yield to too big step sizes, since non-optimal data sets usually include many loose points, which are increasing the step size too far. From here on we tried out different step sizes until we achieved the best coverage, since we did not find a generally applicable criterion.

3.3.3 Using the algorithm on high-dimensional test data

Analogously to chapter 3.2.5, this method also has to be tested in higher dimensions to verify its functionality. For that matter we used the same data sets containing two equally big clusters as in chapter 3.2.5. As mentioned above, for each case we have to try out different step sizes and use the one that has the most amount of data covered and is able to find two approximately equally big clusters.

Table 5: Results for different dimensions.

Dimension	Step size	Voxels	Clusters	Data Covered
2	2	136	2	100%
3	2	565	2	100%
4	3	807	2	100%
5	5	484	2	100%
6	6	633	2	100%

The results are listed in Table 5. We can see, that the algorithm works well on this optimal data set. There is no loss of data in higher dimension as it was the case using spherical voxels, since we covered 100% of the data in every dimension. However, using the clustering process as described in chapter 3.3.1 leads to small losses of the resulting clusters owing to not considering detached voxels without any neighbors. In any case the both detected clusters contained at least 49.7% of the data.

3.3.4 Analyzing the simulated PXD data

In this section we use the algortihm on selected clusters of the simulated PXD data. For the first tries we chose the clusters for pions (PI), anti-deuterons (DD) and tetra quarks (T). The results of the DD and PI cluster are listed in table 6.

Table 6: The results for the PI+DD Cluster

Step size	Clusters	Main Cluster	Biggest secondary Cluster	Total Data covered
10	470	3293 Voxels 59.0% total data	1 Voxel 7.3% total data	77%
15	45	2349 Voxels 90.0% total data	1 Voxel 3.2% total data	100%
20	23	1270 Voxels 79.7% total data	1 Voxels 1.4% total data	86%

For each run we listed the clusters detected by this method, the proportion of total data in the biggest and second biggest cluster and the number of voxels

they consist of aswell as the total data covered. We tried three different step sizes for every combination of two out of the three clusters. As we can see in table 6 the method detects 23 clusters by using a relatively big step size of 20 units. The amount of clusters detected increases the smaller the step size is and is getting up to 430 clusters using the smallest step size of 10 units. As mentioned before, our algorithm only ignores clusters that consist of less than two voxels and contain less than 0.1% of the total data. However there are still many clusters that either consist of two to five voxels only containing a few data points, which causes this big number. These can be ignored aswell and we only pay attention to the clusters containing a perceptible amount of data or voxels. In every case the biggest cluster detected, the main cluster, contains a huge proportion of the total data covered and consists of the most voxels by far. The second biggest cluster only consists of one voxel for every step size, but contains up to 7.3% of the total data, which is about 7540 data points. By decreasing the step size the ratio of total data and data in the main cluster shifts, causing the second biggest cluster containing a bigger proportion of the total data. This could be a hint, that there is a second big accumulation of data besides the main cluster in our data set. Since the computing effort increases exponentially by lowering the step size even more, smaller step sizes could not be tested. Additionally we notice, that we achieved an optimal data coverage using a stepize of 15 units, with having every data point in a voxel. Decreasing or increasing the step size from this value led to a loss of data points. By decreasing the step size even further we assume that the loss of data will get bigger. We repeated the same process on the other two combinations of the data sets, which is DD+T aswell as PI+T. The results are shown in the tables 7 and 8.

Table 7: The results for the DD+T Cluster

Step size	Clusters	Main Cluster	Biggest secondary Cluster	Total Data covered
10	39	3373 Voxels 57.0% total data	1 Voxel 1.3% total data	68.9%
15	23	2239 Voxels 86.1% total data	1 Voxel 2.2% total data	100%
20	14	1580 Voxels 73.4% total data	1 Voxels 2.8% total data	85.0%

Table 8: The results for the PI+T Cluster

Step size	Clusters	Main Cluster	Biggest secondary Cluster	Total Data covered
10	87	4610 Voxels 59.9% total data	1 Voxel 6.3% total data	75.9%
15	45	2349 Voxels 89.99% total data	1 Voxel 3.2% total data	100%
20	23	1494 Voxels 79.5% total data	1 Voxels 1.2% total data	86%

Comparing the results of the algorithm on the different sets of the cluster data, we can see some similarities. For every combination, the step size of 15 units achieved 100% coverage of the total data. The biggest secondary cluster only consists of 1 voxel in every case and contains 7.3% of the total data at most, which means that we can not separate any of the two clusters from each other for sure. It is notable, that the algorithm can find one accumulation of data consisting of a large amount of voxel for every case. The only noteworthy difference between the results of different combinations is the number of clusters deteceted. While the algorithm finds up to 470 clusters for the DD+PI cluster, it only finds 39 clusters for the DD+T clusters. This could be a measure of how tight the data points are, which shows how similar they are to one another. Applying the algorithm to a data set with points very similar to each other would probably lead to less clusters detected by this method, since the voxels created are more likely to be next to each other. To get an idea about how many of the voxels are directly connected to each other, we looked at how many neighbors each voxel have and counted them. The histogram of that for PI+T with a step size of 15 units can bee seen in figure 40. For each dimension one voxel can have two neighbors at most, which would be 12 neighbors for this 6-dimensional data set. As we can see in that histogram, there is no voxel that has a neighbor in every direction. There are only a very few voxels having 8 neighbors, while no voxel has 9 or more. This means, that in at least one direction, every voxel building the main cluster of about 2350 voxels is the border of the detected cluster. Considering voxels that have the highest number of neighbors could help to identify the innermost voxels and therefore the centre of the cluster. We assume that the step sizes used on this data set are too rough to edge the cluster accurately enough, since every voxel borders the cluster. On the other hand there is also a relatively high amount of voxels with no neighbors, which indicates the large spread of the data points in this set. The amounts

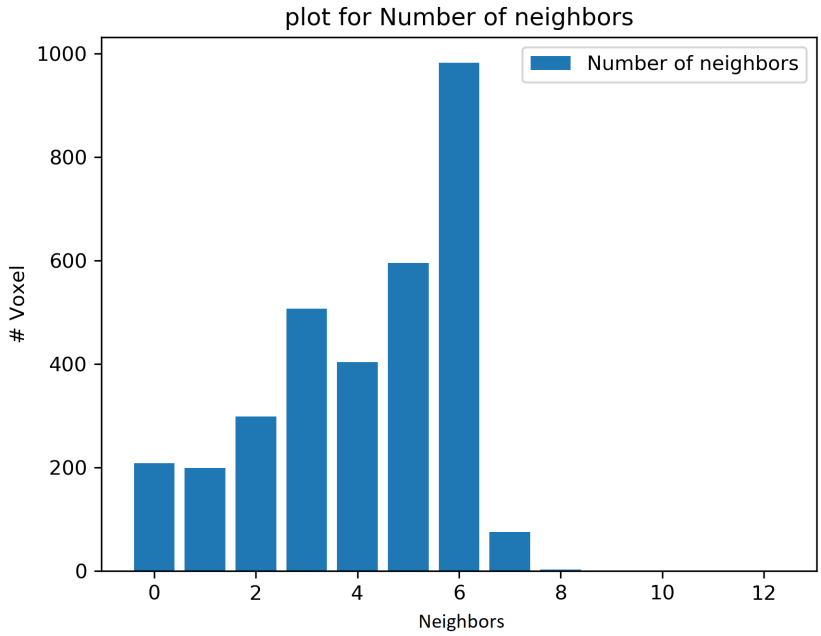

Figure 40: Histogram of the number of neighbors for PI+T

of neighbors, with exception of 4 neighbors, increases steadily and peakes at a voxel having 6 neighbors.

3.4 Conclusion

In this chapter we discussed two approaches in finding the geometry of data sets. Firstly we had used an semi stochastic system of determining the shape of data sets. This was done by placing spheres in a random manner outside of the data set and finding the borders of the data clouds. Than we looked for the larges possible spheres and build up neighbor chains. The second step is the least problematic, but it greatly depends on the first one. The issues with finding the border is two fold, first we need to figure out how many trails we want to run. More trails is always better and we do not run into problems if we increase the number of trails, but time will increase linearly with number of trails. The second problem is, that we need to provide a suitable search radius within which there are no data points before we can start looking for border points. The underlying problem here is, that we do not know which size is appropriate, since the density of data points can vary greatly.

The previously introduced method of using cubical voxels to enclose a data set is only to some extend capable of finding clusters. In contrast of conventional

methods like the Self-organizing-map, this method uses a purely geomtrical approach to find clusters. Therefore the clusters in the given data sets have to be clearly separated without growing into each other at any point. As discussed before, another issue of this method is finding the correct value of the step size. The edges of the clusters to be detected in the data set have to be at least one unit of the step size apart from each other in order to be separated correctly. Choosing smaller step sizes than presented could not be tested in this work, since the computing effort increases dramatically with lowering the step size and we lack of computing capacity to do that.

For the cluster data created by python used to test our method, the algorithm achieved nearly perfect results up to 6 dimensions. Considering that the clusters of the simulated PXD data have a large spread and even common data points, this purely geometrical approach like this method will not work and it will be necessary to find a dynamic approach in determining which voxels are connected than it was laid out in section 3.2.3.

4 Finishing remarks

4.1 Conclusion

This work discussed three different approaches for detecting different point cloud clusters for applying them on the simulated PXD data of the Belle II experiment. The Masters's Thesis from Katharina Dort [1] concerned with this topic, tried to seperate different clusters with Self-organizing-maps, so we started our work with a brief introduction and an own implementation of this form of unsupervised machine learning. Since the previous work was not able to differentiate accurately between more than two clusters of the PXD data, we mainly focussed on geometrical approaches to detect distinct clusters in the PXD data sets. First tests with the Self-organizing-map in this work showed the principle functionality on different test data sets, but also her limits on the PXD data.
The main goal of this work was to investigate the possibility of using voxels to cover the data clusters and use geometrical approaches to differentiate between distinct clusters in the given data set. Two different methods were presented and tested in this work. The first method presented was stochastic and used spherical voxels, while the other one was more systematical and used cubical voxels. It was shown, that both algorithms were able to differentiate between clearly separated clusters in low- and highdimensional test data sets. The stochastic approach however misses data points with increasing the dimensionality, while the systematical approach using cubical voxels did not endure any data losses for higher dimensions. By applying the methods to the simulated PXD data the weaknesses of these methods were shown clearly. Both our two approaches relied strongly on an input parameter that was related to the cloud density or the mean distance between two points. Furthermore the data clouds of PI, DD, T and BG from the simulated data sets are partially overlapping, which means that this purely geometrical methods will not work without further modification. The current methods however are already applicable and functional for detecting clearly separated clusters and can act as an assistance or alternative for unsupervised learning methods, since it is able to identify the correct amount of clusters and the data they consist of without any information of the data set.

4.2 Outlook

As mentioned before, the methods presentend in this work are purely geometrical. So a simple check if two voxels touch each other fails at differentiating between two clusters if the data clouds overlap. In section 3.2.7 we noticed that taking the midpoint norms into account could help telling clusters apart. Besides that further considerations could be taking in to account for the clustering process. For the case that clusters overlap at some points, it could be instructive to look at the density gradient of the voxels, meaning how many data points are inside a voxel per volume. Thus one could make a decision based on a varying density, since we assume that the part where the clusters overlap are less dense than their centers. Continuing improvements on that algorithm could not

only consider direct, but also further neighbors. Central voxels are more likely to have a constant quotient of direct-to-total number of neighbors, while we that assume voxels connecting two clouds have a small quotient. Further works on this matter could help detecting those kind of voxels that are less dense and connect two different clusters and lead to a more sophisticated clustering algorithm.

5 Appendix

5.1 SOM Python Code

```
# Module importieren
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
import math
import argparse
import random
import time
from progress.bar import ChargingBar
# selbst geschriebene Helfer importieren
# "helper" enth lt s mtliche Funktionen die w hrend des Programm
    gebraucht werden
# "plotter" wird gebraucht um die Graphen zu plotten
from helper import *
from plotter import *

# schaltet die Warnung aus, dass zu viele Graphen geplotet wurden
matplotlib.rc('figure', max_open_warning = 0)

# input flags werde hier geparst
parser = argparse.ArgumentParser()
parser.add_argument("-v", "--verbosity", help="output verbosity",
    action="count")
parser.add_argument("-p", "--plots", help="output plots, {cluster,
    movement, norms, neighbor}", type = str)
parser.add_argument("-d", "--data", help="determines whicht data
    set is beging used {dd, pi, t, test}", type = str)
parser.add_argument("-e", "--elements", help="the number of
    datapoints to be processed", type = int)
parser.add_argument("-w", "--weight", help="sets the weight of node
    adjustment", type = float)
parser.add_argument("-c", "--cutoff", help="sets the cutoff radius"
    , type = int)
args = parser.parse_args()

if args.data != None:
    dataSplit = args.data.split('+')

if args.plots != None:
    plotsSplit = args.plots.split('+')

# user bestimmter prefix f r plots
plotPrefix = input("Enter prefix for output files ")

# startup Parameter einstellen
# "gewicht" gibt die Gewichtung mit der Nachbarknoten angepasst
    werden
# "cutoff" gibt die distance an bis zu welchem Nachbarn Knoten
    angepasst werden
sigma= 0.5

# output f r die input flags
if args.weight == None:
```

```
    gewicht = 2
    print("Using default weight of value 2")
else:
    gewicht = args.weight
    print("Using a weight of value {}".format(args.weight))
if args.cutoff == None:
    cutoff = cutneighborhood(2)
    args.cutoff=2
    print("Using default cutoff radius of 2")
else:
    cutoff = cutneighborhood(args.cutoff)
    print("Using a cutoff radius of {}".format(cutoff))

# Daten einlesen und in Array konvertieren
setDD = False
setPI = False
setT = False
setBG = False

startReading = time.time()
if args.data != None:
    dataSplit = args.data.split('+')
    if ("test" in dataSplit):
        #data = genData(6,3,24000)
        #data = genDataNew(24000)
        data = sixDData(24000 ,15)
    else:
        if ("bg" in dataSplit):
            fileBG = "Traindata/Beam_BG_cluster_sim.txt"
            dataBG = np.genfromtxt(fileBG, delimiter=' ')
            setBG = True
        if ("dd" in dataSplit):
            fileDD = "Traindata/Beam_dd_cluster_sim.txt"
            dataDD = np.genfromtxt(fileDD, delimiter=' ')
            setDD = True
        if ("pi" in dataSplit):
            filePI = "Traindata/Beam_pi_cluster_sim.txt"
            dataPI = np.genfromtxt(filePI, delimiter=' ')
            setPI = True
        if ("t" in dataSplit):
            fileT = "Traindata/Beam_T_cluster_sim.txt"
            dataT = np.genfromtxt(fileT, delimiter=' ')
            setT = True

        if setDD == True and setPI == True and setT == True and
    setBG == True:
            data = np.vstack((dataDD, dataPI, dataT, dataBG))
        elif setDD == False and setPI == True and setT == True and
    setBG == True:
            data = np.vstack((dataPI, dataT, dataBG))
        elif setDD == True and setPI == False and setT == True and
    setBG == True:
            data = np.vstack((dataDD, dataT, dataBG))
        elif setDD == True and setPI == True and setT == False and
    setBG == True:
            data = np.vstack((dataDD, dataPI, dataBG))
```

```
97          elif setDD == False and setPI == False and setT == True and
     setBG == True:
98              data = np.vstack((dataT, dataBG))
99          elif setDD == False and setPI == True and setT == False and
     setBG == True:
100             data = np.vstack((dataPI, dataBG))
101         elif setDD == True and setPI == False and setT == False and
     setBG == True:
102             data = np.vstack((dataDD, dataBG))
103         elif setDD == True and setPI == True and setT == True and
    setBG == False:
104             data = np.vstack((dataDD, dataPI, dataT))
105         elif setDD == False and setPI == True and setT == True and
    setBG == False:
106             data = np.vstack((dataPI, dataT))
107         elif setDD == True and setPI == False and setT == True and
    setBG == False:
108             data = np.vstack((dataDD, dataT))
109         elif setDD == True and setPI == True and setT == False and
    setBG == False:
110             data = np.vstack((dataDD, dataPI))
111         elif setDD == False and setPI == False and setT == True and
     setBG == False:
112             data = dataT
113         elif setDD == False and setPI == True and setT == False and
     setBG == False:
114             data = dataPI
115         elif setDD == True and setPI == False and setT == False and
     setBG == False:
116             data = dataDD
117
118 if args.data == None:
119     fileBG = "Traindata/Beam_BG_cluster_sim.txt"
120     dataBG = np.genfromtxt(fileBG, delimiter=' ')
121     fileDD = "Traindata/Beam_dd_cluster_sim.txt"
122     dataDD = np.genfromtxt(fileDD, delimiter=' ')
123     data = np.vstack((dataDD, dataBG))
124
125 np.random.shuffle(data)
126 endReading = time.time()
127
128
129 # Meta Daten bestimmen und Knoten generieren
130 # "getMetrics" gibt das gr   te & kleinste Element pro Dimension
    aus
131 # "vecSize" ist die Dimension jedes Datenpunkts
132 # "numElements" ist die Anzahl an Datenpunkten
133 # "genNodes" generiert die Knoten, jedem Knoten werden zuf llige
    Wert in
134 # Abh ngigkeit der maximalen & minmalen Werte pro Dimension
    generiert
135 # "nodes" ist ein [nx    ny    vecSize] Array
136 minElement, maxElement, vecSize, numElements = getMetrics(data)
137
138 if args.elements == None:
139     print("All {} data points will be processed".format(numElements
    ))
```

```
elif args.elements > numElements:
    print("Input is bigger than availible data points.")
    print("This means all {} data points will be processed".format(
    numElements))
else:
    numElements = args.elements
    print("{} elements will be processed".format(args.elements))

# Neuronenanzahl bestimmen
neurons = 15
print("{} times {}".format(neurons,neurons))
nodes = genNodes(neurons, vecSize, minElement, maxElement, False)

outPutCollection = []
learningPlot = []
processedPoints = 0
startProcess = time.time()
# hier startet der Loop  ber  die Daten, "step" ist der Zeitschritt
     in data
if args.verbosity == None:
    bar = ChargingBar(" ", max=numElements)
for timeStep in range(0,numElements):
    # normList ist eine Liste der Normen eines Datenpunkts gegen
    jeden Knoten
    normList = []
    normList = normListe(nodes, data[timeStep], neurons)
    # hier wird der Index des Knoten mit der kleinsten Norm
    bestimmt
    minIndex = winNode(normList)
    outPut = [timeStep, minIndex[0], minIndex[1]]
    outPut = np.append([outPut],[data[timeStep]])
    if timeStep != 0:
        outPutCollection = np.vstack((outPutCollection, outPut))
    else:
        outPutCollection = np.append(outPutCollection, outPut)
    # anpassen des Vektors des Knotens der kleinsten Norm
    # "adjustNode" nimmt einen Vektor, einen Datenpunkt und ein
    Gewicht an
    nodes[minIndex] = adjustNode(nodes[minIndex], data[timeStep],
    timeStep, gewicht)
    UMatrix=genUMatrix(nodes, neurons)
    learnRate = learningRate(timeStep, gewicht)
    learningPlot = np.append(learningPlot, learnRate)
    if args.verbosity == 1:
        print("Time Step:", timeStep, "\n", "Winning Node Index:",
    minIndex, "\n")
    elif args.verbosity == 2:
        print("Time Step:", timeStep, "\n", "Winning Node Index:",
    minIndex, "\n", "Learningrate:", learnRate, "\n")
    else:
        pass
        #print("Time Step:", timeStep)
    nodes, neighList = adjustNeighbor(nodes, data[timeStep],
    minIndex, neurons, neurons, timeStep, gewicht, cutoff, args.
    cutoff)
    normListAfter = []
    normListAfter = normListe(nodes, data[timeStep], neurons)
```

```
187     # plotten der Daten
188     if processedPoints % 2000 == 0:
189         if args.plots == None:
190             startPlot = time.time()
191             plotNodes(timeStep, normList-normListAfter, "Movement",
     plotPrefix)
192             plotNodes(timeStep, UMatrix, "Cluster", plotPrefix)
193         else:
194             startPlot = time.time()
195             if ("all" in plotsSplit) == True:
196                 plotNormList(timeStep, minIndex, normList,
    normListAfter, plotPrefix)
197                 plotNodes(timeStep, 1/neighList, "Neighborhood",
    plotPrefix)
198                 plotNodes(timeStep, normList-normListAfter, "
    Movement", plotPrefix)
199                 plotNodes(timeStep, UMatrix, "Cluster", plotPrefix)
200             if ("norm" in plotsSplit) == True:
201                 plotNormList(timeStep, minIndex, normList,
    normListAfter, plotPrefix)
202             if ("movement" in plotsSplit) == True:
203                 plotNodes(timeStep, normList-normListAfter, "
    Movement", plotPrefix)
204             if ("cluster" in plotsSplit) == True:
205                 plotNodes(timeStep, UMatrix, "Cluster", plotPrefix)
206             if ("neighbor" in plotsSplit) == True:
207                 plotNodes(timeStep, 1/neighList, "Neighborhood",
    plotPrefix)
208             if ("none" in plotsSplit) == True:
209                 pass
210     processedPoints += 1
211     if 'bar' in locals():
212         bar.next()
213 if 'bar' in locals():
214     bar.finish()
215 endPlot = time.time()
216 endProcess = time.time()
217
218 # learnRate plotten
219 fig = plt.figure()
220 plt.plot(learningPlot)
221 plt.savefig("Output/learnRate_{}.png".format(plotPrefix))
222
223 # Schreiben der output Daten
224 np.savetxt("Output/umatrix_{}.txt".format(plotPrefix), UMatrix, fmt
    ="%f")
225 np.savetxt("Output/finalData_{}.txt".format(plotPrefix),
    outPutCollection, fmt="%f")
226
227 # Abschluss Kommentar
228 if processedPoints == numElements:
229     print("Run was successful")
230     print("Reading the data took {}".format(endReading-startReading
    ))
231     print("The main process took {}".format(endProcess-startProcess
    ))
```

```
    print("That is {} per data point".format((endProcess-
    startProcess)/numElements))
    print("The plots took {}".format(abs((endPlot-startPlot)-(
    endProcess-startProcess))*numElements))
else:
    print("The programm crashed at data point {}".format(
    processedPoints))
```

Listing 1: Main Program

```
# das ist die "helper" Datei, die s mtiche funktionen enth lt
import numpy as np
import math
import random

# diese Funktion generiert die Knoten und initialisiert zuf llige
    Neuronen
def genNodes(neurons: int, vecSize: int, min: int, max: int,
    randNodes = True):
    if randNodes == True:
        nodes = np.random.rand(neurons,neurons,vecSize) #
    vollst ndig zuf llig
    else:
        nodes = np.random.randint(min,max,size=(neurons,neurons,
    vecSize)) # basierend auf Input Datan
        nodes = np.array(nodes, dtype='float')
    return nodes

# schnelle Funktion zur Bestimmung der Norm
def fastNorm(vec):
    return math.sqrt(np.dot(vec, vec.T))

# auslesen der Meta Datan des Inputs
def getMetrics(data: np.array):
    maxData = data.max(axis=0) # gr   te Daten im Input
    maxElement = np.ceil(maxData) # runden der max Werte des Inputs
    minData = data.min(axis=0) # kleinste Daten im Input
    minElement = np.floor(minData) # runden der minmal Werte des
    Inputs
    vecSize = np.size(data, axis=1) # Gr   e des Vektors
    numElements = np.size(data, axis=0) # Anzahl an Elementen
    return (minElement, maxElement, vecSize, numElements)

def numNeurons(numElements: int):
    neurons = np.ceil(math.sqrt(5*math.sqrt(numElements)))
    neurons = neurons.astype(int)
    if neurons < 10:
        neurons = 10
    return neurons

# erstellen der Abst nde zwischen einem Datenpunkt und den
    Neuronen
def normListe(nodes: np.array, data: np.array, neurons: int):
    normList = []
    for Elements in nodes:
        for Element in Elements:
            diff = fastNorm(data-Element)
            normList = np.append(normList, diff)
```

```
43      return normList.reshape(neurons,neurons)
44
45  # findet heraus welcher Knoten den geringsten Abstand zum
        Datanpunkt hat
46  def winNode(normList: np.array):
47      minIndex = np.where(normList == normList.min())
48      return (minIndex[0][0],minIndex[1][0])
49
50  # wie schnell die Lernrate abf llt
51  def learningRate(timeStep: int, gewicht: int):
52      return 1/(math.log(timeStep+2)*gewicht)
53
54  # Funktion zum verschieben eines Neurons zum Datanpunkt hin
55  def adjustNode(node: np.array, data: np.array, timeStep: int,
        neighCoeff: int):
56      node += (data - node)*learningRate(timeStep, neighCoeff)
57      return node
58
59  # eine Funktion zum anpassen der Nachbarknoten des gewinnenden
        Knotens
60  def adjustNeighbor(nodes: np.array, data, minIndex: tuple, nx: int,
         ny: int, timeStep: int, gewicht: float, cutoff: float,
        neighcut: int):
61      neighList=[]
62      for i in range(0,nx):
63          neighCoeff = gewicht
64          for j in range(0,ny):
65            if (math.sqrt((i-minIndex[0])**2+(j-minIndex[1])**2)) <=
        cutoff and ((i-minIndex[0]) <= neighcut) and ((j-minIndex[1])
        <= neighcut):
66              #print(i,j)
67              neighCoeff = gewicht
68              if (i != minIndex[0]) and (j != minIndex[1]):
69                  neighCoeff=neighCoeff*2**(abs(i-minIndex[0])+abs(j-
        minIndex[1]))
70                  #print("Neighborhood coeff:", neighCoeff, "\n")
71                  neighList=np.append(neighList, neighCoeff)
72                  nodes[i,j]=adjustNode(nodes[i,j], data, timeStep,
        neighCoeff)
73
74              if ((i != minIndex[0]) and (j == minIndex[1])) or (i ==
         minIndex[0]) and (j != minIndex[1]) :
75                  neighCoeff=neighCoeff*2**(abs(i-minIndex[0])+abs(j-
        minIndex[1]))+abs(i-minIndex[0])+abs(j-minIndex[1])
76                  #print("Neighborhood coeff:", neighCoeff, "\n")
77                  neighList=np.append(neighList, neighCoeff)
78                  nodes[i,j]=adjustNode(nodes[i,j], data, timeStep,
        neighCoeff)
79
80              if (j == minIndex[1]) and (i == minIndex[0]):
81                  neighList=np.append(neighList, gewicht)
82            else:
83                neighList=np.append(neighList, 9999)
84
85      neighList = np.reshape(neighList,(nx,ny))
86      #print(neighList)
87
```

```
88      return (nodes, neighList)
89
90  def GaussNeighbor(nodes: np.array, data, minIndex: tuple, nx: int,
        ny: int, timeStep: int, gewicht: float, cutoff: float, neighcut
        : int, sigma: float):
91      neighList=[]
92      for i in range(0,nx):
93          for j in range(0,ny):
94            if (math.sqrt((i-minIndex[0])**2+(j-minIndex[1])**2)) <=
        cutoff and (math.sqrt((i-minIndex[0])**2) <= neighcut) and (
        math.sqrt((j-minIndex[1])**2) <= neighcut):
95              if (j!=minIndex[1]) or (i!=minIndex[0]):
96                  neighCoeff = math.exp((i-minIndex[0])**2+(j-
        minIndex[1])**2/(2*sigma))
97                  neighList=np.append(neighList, neighCoeff)
98                  nodes[i,j]=adjustNode(nodes[i,j], data, timeStep,
        neighCoeff)
99
100             else:
101                 neighCoeff= gewicht
102                 neighList=np.append(neighList, neighCoeff)
103
104           else:
105               neighList=np.append(neighList, 9999)
106
107
108
109     neighList = np.reshape(neighList,(nx,ny))
110     #print(neighList)
111
112     return (nodes, neighList)
113
114
115 # generiert zuf llige, nicht zusammenh ngende Daten zum testen
        des Programms
116 def genData(vecDim: int, numCubes: int, numElements: int):
117     testData = []
118     for cubeNumber in range(0,numCubes):
119         if cubeNumber % 2 == 0:
120             max = 100*cubeNumber+95
121             min = 100*cubeNumber+5
122         else:
123             max = 100*cubeNumber+90
124             min = 100*cubeNumber+10
125         elPerCube = np.round(numElements/numCubes,0).astype(int)
126         dataPoint = np.random.randint(min,max,size=(elPerCube,
        vecDim))
127         dataPoint = np.array(dataPoint/10, dtype='float')
128         testData = np.append(testData, dataPoint)
129     testData = testData.reshape(numElements,vecDim)
130     np.random.shuffle(testData)
131     return testData
132
133 def genDataNew(number):
134     n = np.round(number/6,0).astype(int)
135     a = np.random.multivariate_normal([0, 5], [[2, 1], [1, 1.5]],
        size=n)
```

```
136     b = np.random.multivariate_normal([2, 0], [[1, -1], [-1, 3]],
        size=n)
137     c = np.random.multivariate_normal([5, 4], [[5, 0], [0, 1.2]],
        size=n)
138     d = np.random.multivariate_normal([14, 18], [[5, 2], [2, 4]],
        size=n)
139     e = np.random.multivariate_normal([19, 15], [[2, -2], [-2, 6]],
         size=n)
140     f = np.random.multivariate_normal([19, 18], [[4, 0], [0, 1]],
        size=n)
141     z = np.concatenate((a, b, c, d, e, f))
142     return z
143
144 # die UMatrix aus den vorliegenden daten bestimmen, sie zeigt
        sp ter cluster
145 def genUMatrix(nodes: np.array, neurons: int) :
146     UMatrix=[]
147     for i in range(0, neurons):
148         for j in range(0,neurons):
149             distance=0
150             nrneighbour=0
151             if (j+1 % neurons != 0) and (j != neurons-1):
152                 distance+=fastNorm(nodes[i][j+1]-nodes[i][j])
153                 nrneighbour+=1
154             if j % neurons != 0:
155                 distance+=fastNorm(nodes[i][j-1]-nodes[i][j])
156                 nrneighbour+=1
157             if i+1 < neurons:
158                 distance+=fastNorm(nodes[i+1][j]-nodes[i][j])
159                 nrneighbour+=1
160             if i-1 >= 0:
161                 distance+=fastNorm(nodes[i-1][j]-nodes[i][j])
162                 nrneighbour+=1
163             Udistance=distance/nrneighbour
164             UMatrix=np.append(UMatrix,Udistance)
165     UMatrix = UMatrix.reshape(neurons,neurons)
166     return UMatrix
167
168 # eine Funktion zur Bestimmung des "cutoff" Radius'
169 def cutneighborhood(cutoff: int):
170     return math.sqrt(2*cutoff**2)
171
172 # Sph ren werden hier definiert, sie besitzen einen Radius, einen
        Mittelpunkt
173 # dieser wird durch die dimensionalit t der Daten bestimmt, nicht
        hier, aber in einer Funktion
174 # "number" soll die Sp hren durchz hlen und "counter" die
        Datenpunkte, die innerhalb der Sph re liegen
175 # Sph ren k nnen wachsen und man kann die Z hler heraufsetzen
176 class sphere:
177     def __init__(self, number: int, radius: float, midPoint: np.
        array, counter: int):
178         self.number = number
179         self.radius = radius
180         self.midPoint = np.array([midPoint])
181         self.counter = counter
182     def getRadius(self):
```

```
        return self.radius
    def grow(self, speed: float):
        self.radius = self.radius*speed
    def addCounter(self):
        self.counter += 1
    def moveMidPoint(self, speed: float, direction: np.array):
        self.midPoint += speed*direction

# nimmt einen Vektor entgegen und    berprft    ob dessen Norm
    gr   er  oder kleiner ist als eine vorgegeben Zahl
def checkCollision(distanceVec: np.array, radius: float):
    if fastNorm(distanceVec) > radius:
        return False
    return True

# generiert Sph ren aus einem Datensatz, als Mittelpunkt wird ein
    zuf lliger Messwert genommen
def createCloud(number:int, radius: float, data: np.array, counter:
     int):
    midPoint = random.choice(data)
    cloud = sphere(number, radius, midPoint, counter)
    return cloud

class clusters:
    def __init__(self, point: np.array, timeStep: int, index: tuple
    ):
        self.point = np.array([point])
        self.timeStep = timeStep
        self.index = index


def sixDData(number = 1800, spread = 50):
    n = np.round(number/2,0).astype(int)
    a = np.random.multivariate_normal([24, 0, 0, 0, 0, 0], [[spread
    , 0, 0, 0, 0, 0], [0, 1.95*spread, 0, 0, 0, 0], [0, 0, spread,
    0, 0, 0], [0, 0 ,0, 1.95*spread, 0, 0], [0, 0, 0, 0, spread,
    0], [0, 0 ,0, 0, 0, 1.95*spread]], size=n)
    b = np.random.multivariate_normal([0, 0, 0, 0, 0, 0], [[spread,
     0, 0, 0, 0, 0], [0, spread, 0, 0, 0, 0], [0, 0, 1.95*spread,
    0, 0, 0], [0, 0 ,0, 1.95*spread, 0, 0], [0, 0, 0, 0, spread,
    0], [0, 0 ,0, 0, 0, 1.95*spread]], size=n)
    z = np.concatenate((a, b))
    return z
```

Listing 2: Code including the functions used in the main Program

```
# hier sind alle Plot-Funktionen
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg as la

# plottet die Abst nde zwischen einem Datenpunkt und den Neuronen,
     vor und nach Anpassung eines Knotens und nach der
    Nachbarschaftsanpassung
def plotNormList(timeStep: int, minIndex: tuple, normList: np.array
    , normListAfter: np.array, prefix: str):
    fig = plt.figure(figsize=(6,9))
    # Plot vor Anpassung
```

```
    plt.subplot(2,1,1)
    plt.imshow(normList.T, origin='lower', interpolation='nearest')
    plt.scatter(minIndex[0],minIndex[1], color="red", marker=".")
    plt.ylabel("y-axis", fontsize=12)
    cbar = plt.colorbar()
    cbar.ax.get_yaxis().labelpad = 15
    cbar.ax.set_ylabel('distance', rotation=270, fontsize = 12)

    # Plot nach Anpassung
    plt.subplot(2,1,2)
    plt.imshow(normListAfter.T, origin='lower', interpolation='
    nearest')
    plt.scatter(minIndex[0],minIndex[1], color="red", marker=".")
    title = "norm plot of {} at time step: {}".format(minIndex,
    timeStep)
    fig.suptitle(title, fontsize=16)
    plt.xlabel("x-axis", fontsize=12)
    plt.ylabel("y-axis", fontsize=12)
    cbar = plt.colorbar()
    cbar.ax.get_yaxis().labelpad = 15
    cbar.ax.set_ylabel('distance', rotation=270, fontsize = 12)

    plt.savefig("Norms/plot_{}_timeStep_{}.png".format(prefix,
    timeStep), dpi=300)

def plotNodes(timeStep: int, array: np.array, name: str, prefix = '
    '):
    fig = plt.figure()
    plt.imshow(array, origin='lower', interpolation='nearest')
    title = "{} Plot at {}".format(name,timeStep)
    fig.suptitle(title, fontsize=16)
    plt.xlabel("x-axis", fontsize=12)
    plt.ylabel("y-axis", fontsize=12)
    axes = plt.gca()
    cbar = plt.colorbar()
    cbar.ax.get_yaxis().labelpad = 15
    cbar.ax.set_ylabel('value', rotation=270, fontsize = 12)
    plt.savefig("{}/plot_{}_timeStep_{}.png".format(name, prefix,
    timeStep), dpi=300)
```

Listing 3: Plot functions for the SOM

5.2 Stochastic approach

```
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import random
import math
import copy
import time
import itertools
from progress.bar import ChargingBar
from scipy.spatial.distance import pdist
from scipy.spatial.distance import cdist
from VoxStochHelper import *
from clusterGenerator import *
```

```
from extraHelper import *
#np.random.seed(0)

whichData = "pi+dd+t"
startStart = time.time()
# daten erstellen
start = time.time()
testData = dataGenerator(whichData)
minElement, maxElement, meanElement, vecSize, numElements =
    getMetrics(testData)
end = time.time()
readingData = end-start
print("generating data and metrics took {0:.4f} seconds.".format(
    readingData))
print("")


# rand finden
start = time.time()
searchRadiusMax, searchRadiusMean, searchRadiusMin =
    findSearchRadius(testData)
searchRadius = closestNumber((searchRadiusMax + searchRadiusMean)
    /2,0.25)
trailData = np.unique(testData, axis=0)
boundIndex = findBound(trailData, minElement, maxElement, trails =
    120000, searchRadius = searchRadius)
boundData, restData = splitData(trailData, boundIndex)
midPointPool = np.unique(restData, axis=0)
end = time.time()
findingBound = end-start
print("finding the boundary with a searchRadius of {0} took {1:.4}
    seconds.".format(searchRadius ,findingBound))
print("")
print("there are {} data points of which {} belong to the boundary.
    ".format(len(testData),len(boundData)))
print("the largest element in each dimension is {} and the smallest
     is {}.".format(maxElement, minElement))
print("the data set has a dimension of {}.".format(vecSize))
print("")


# voxel erstellen
print("creating voxels...")
mainStart = time.time()
voxelNumber = 0
voxels = np.array([])
counter = 0
stop = False
rejectedVoxels = 0
while stop == False:
    oneNorm = np.array([])
    start = time.time()
    length = len(midPointPool)
    print("looking for the largest voxel possible...")
    bar = ChargingBar(" ", max=length)
    for point in midPointPool:
        _, normBound = closestDistance(point, boundData)
```

```
63
64          if voxels.size != 0:
65              voxNorm = np.array([])
66              for vox in voxels:
67                  norm = fastDiff(point, vox.midPoint) - vox.radius
68                  voxNorm = np.append(voxNorm, norm)
69              stepNorm = voxNorm.min()
70
71          if 'stepNorm' in locals():
72              if stepNorm < normBound:
73                  oneNorm = np.append(oneNorm, stepNorm)
74              else:
75                  oneNorm = np.append(oneNorm, normBound)
76          else:
77              oneNorm = np.append(oneNorm, normBound)
78          bar.next()
79      bar.finish()
80      end = time.time()
81      print("finding midPoint and radius took {:.4} seconds.".format(
        end-start))
82
83      radius = oneNorm.max()
84      index = np.argmax(oneNorm)
85
86      midPoint = midPointPool[index]
87      print("creating a voxel number {} at {} with radius {}.".format
        (counter ,midPoint, radius))
88      voxel = spherus(counter, midPoint, radius)
89
90      if len(voxels) > 0:
91          voxIndex = findNextVoxel(voxel.midPoint, voxels,len(voxels)
        )[0]
92          distanceToNextVoxel = fastNorm(voxel.midPoint-voxels[
        voxIndex].midPoint) - voxels[voxIndex].radius
93          voxNorm = fastNorm(voxel.midPoint-voxels[voxIndex].midPoint
        ) - (voxels[voxIndex].radius + voxel.radius)
94          if voxNorm < searchRadius:
95              voxel.radius = distanceToNextVoxel
96
97      #indices = voxel.appendData(restData, vecSize)
98      #boundIndices = voxel.appendData(boundData, vecSize)
99      norms = cdist([voxel.midPoint],midPointPool).ravel() - voxel.
        radius
100     indices = np.where(norms<=0)
101     #print(np.shape(indices)[1])
102     #print("it contains {0} data points and has a desnsity of {1:.4
        f}.".format(len(voxel.contData),voxel.density))
103     if (np.shape(indices)[1] > 1):
104         print("")
105         voxels = np.append(voxels, voxel)
106         counter += 1
107         #restData = np.delete(restData, indices, axis=0)
108         #boundData = np.delete(boundData, boundIndices, axis=0)
109         midPointPool = np.delete(midPointPool, indices, axis=0)
110         print("")
111     if np.shape(indices)[1] == 1:
112         stop = True
```

```
        rejectedVoxels += 1
        print("voxel rejected and stopping routine.")

end = time.time()
creatingVoxel = end-mainStart
print("")
print("creating voxels took {0:.4} seconds.".format(creatingVoxel))
voxelCount = len(voxels)
print("there are {} voxels.".format(voxelCount))

radiusesOne = np.array([])
densitiesOne = np.array([])
countsOne = np.array([])
normsOne = np.array([])
numOfNeighborsOne = np.array([])

for i in range(0,len(voxels)):
    voxels[i].neighbors = np.append(voxels[i].neighbors, voxels[i].
    voxelNumber)
    for j in range(0,len(voxels)):
        if i != j:
            voxels[i].findNeighbors(voxels[j].midPoint, voxels[j].
    radius, voxels[j].voxelNumber, searchRadius)

for voxel in voxels:
    indices = voxel.appendData(testData, vecSize)
    testData = np.delete(testData, indices, axis=0)
    radiusesOne = np.append(radiusesOne, voxel.radius)
    densitiesOne = np.append(densitiesOne, voxel.density)
    countsOne = np.append(countsOne, len(voxel.contData)).astype(
    int)
    normsOne = np.append(normsOne, fastNorm(voxel.midPoint))
    numOfNeighborsOne = np.append(numOfNeighborsOne, len(voxel.
    neighbors)).astype(int)

voxelsUptoNow = len(voxels)
meanRadius = radiusesOne.mean()

print("{} data points have been covered".format(countsOne.sum()))
print("{0:.2f}% of the data have been covered".format( (countsOne.
    sum()/numElements)*100 ))
print("")


# neue Voxel erstellen
print("growing new voxels...")
stop = False
mainStart = time.time()
lenVox = len(voxels)
coveragePoint = np.round(lenVox*vecSize,0).astype(int)
counterNewVoxels = 0
bar = ChargingBar(" ", max=coveragePoint)
while stop == False:
    pickIndex = np.random.choice(len(midPointPool),1)[0]
    pick = midPointPool[pickIndex]
    pickCopy = copy.copy(pick)
    voxIndex = findNextVoxel(pick, voxels,len(voxels))[0]
```

```
    nextVoxIndex = findNextVoxel(pick, voxels,len(voxels))[1]
    distance = fastNorm(pick-voxels[voxIndex].midPoint) - voxels[
    voxIndex].radius
    distanceCopy = copy.copy(distance)
    nextDistance = fastNorm(pickCopy-voxels[nextVoxIndex].midPoint)
     - (voxels[nextVoxIndex].radius+distanceCopy)
    voxel = spherus(counter, pick, distance)

    speed = 0.001
    grownUp = False
    while grownUp == False:
        # solange wachsen lassen, bis der mittelpunkt und der
    n chste randpunkt gr   er wird
        # wachsen lassen, solange das neue voxel dem nextVoxIndex
    n her kommt?
        voxel.moveMidPoint(speed, voxels[voxIndex].midPoint)
        voxel.growRadius(voxels[voxIndex].midPoint, voxels[voxIndex
    ].radius)
        speed += 0.001
        newDistance = fastNorm(voxel.midPoint-voxels[nextVoxIndex].
    midPoint) - (voxels[nextVoxIndex].radius+voxel.radius)
        nextVoxIndex = findNextVoxel(voxel.midPoint, voxels,len(
    voxels))[1]
        index, boundNorm = closestDistance(voxel.midPoint,
    boundData)
        if newDistance > nextDistance or boundNorm <= voxel.radius:
            grownUp = True
        if checkCollision(voxel.midPoint-voxels[nextVoxIndex].
    midPoint, voxel.radius+voxels[nextVoxIndex].radius) or voxel.
    radius > meanRadius:
            grownUp = True
    #indices = voxel.appendData(restData, vecSize)
    #boundIndices = voxel.appendData(boundData, vecSize)
    norms = cdist([voxel.midPoint],midPointPool).ravel() - voxel.
    radius
    indices = np.where(norms<=0)

    if (np.shape(indices)[1] > 1):
        voxels = np.append(voxels, voxel)
        counter += 1
        counterNewVoxels += 1
        #restData = np.delete(restData, indices, axis=0)
        #boundData = np.delete(boundData, boundIndices, axis=0)
        midPointPool = np.delete(midPointPool, indices, axis=0)
        bar.next()
    else:
        rejectedVoxels += 1
    if counterNewVoxels == coveragePoint:
        stop = True
        print("")
        print("reached coverage point and stopping routine.")
        bar.finish()
    if counterNewVoxels > coveragePoint/2:
        if np.array_equal(voxels[-1].midPoint, voxel.midPoint):
            stop = True

bar.finish()
```

```
211
212 for voxel in voxels:
213     if len(voxel.contData) == 0:
214         indices = voxel.appendData(testData, vecSize)
215         testData = np.delete(testData, indices, axis=0)
216
217 end = time.time()
218 newVoxels = end-mainStart
219 print("creating new voxels took {0:.4} seconds.".format(newVoxels))
220 print("{} new voxels were created.".format(counterNewVoxels))
221 print("")
222
223
224 # cluster finden
225 start = time.time()
226 print("looking for clusters...")
227 bar = ChargingBar(" ", max=7)
228 clusters = []
229 for i in range(0,len(voxels)):
230     for j in range(0,len(voxels)):
231         if i != j:
232             voxels[i].findNeighbors(voxels[j].midPoint, voxels[j].
    radius, voxels[j].voxelNumber, searchRadius)
233     voxels[i].neighbors = np.unique(voxels[i].neighbors)
234     voxels[i].neighbors = np.sort(voxels[i].neighbors, kind='
    mergesort')
235     clusters.append(voxels[i].neighbors)
236 clusters = np.array(clusters)
237 bar.next()
238
239 tuples = list(itertools.permutations(range(len(clusters)), 2))
240 for i,j in tuples:
241     if len(set(clusters[i]).intersection(clusters[j])) != 0:
242         clusters[i] = np.append(clusters[i], clusters[j])
243         clusters[i] = np.unique(clusters[i])
244         clusters[i] = np.sort(clusters[i], kind='mergesort')
245 bar.next()
246
247 indices = np.array([])
248 for i,j in tuples:
249     if np.array_equal(clusters[i],clusters[j]) == True and i not in
     indices:
250         if len(clusters[i]) >= len(clusters[j]):
251             indices = np.append(indices, j).astype(int)
252         if len(clusters[i]) < len(clusters[j]):
253             indices = np.append(indices, i).astype(int)
254 indices = np.unique(indices)
255 hope = np.delete(clusters, indices)
256 bar.next()
257
258 clustersData = {}
259 for i in range(0,len(hope)):
260     iClusterData = np.array([])
261     for j in range(0,len(hope[i])):
262         iClusterData = np.append(iClusterData, voxels[hope[i][j]].
    contData)
263         vertSize = np.round(len(iClusterData)/vecSize,0).astype(int
```

```
    )
264         iClusterData = iClusterData.reshape(vertSize,vecSize)
265     clustersData[i] = iClusterData
266 bar.next()
267
268 clusterMidPoints = {}
269 largestCount = {}
270 smallestCount = {}
271 maxCountIndex = {}
272 for i in range(0,len(hope)):
273     clusterMidPoint = np.zeros(vecSize)
274     dataClusterCount = np.array([])
275     for j in range(0,len(hope[i])):
276         clusterMidPoint += len(voxels[hope[i][j]].contData)*voxels[
    hope[i][j]].midPoint
277         dataClusterCount = np.append(dataClusterCount, len(voxels[
    hope[i][j]].contData))
278     largestCount[i] = dataClusterCount.max()
279     maxCountIndex[i] = np.argmax(dataClusterCount)
280     smallestCount[i] = dataClusterCount.min()
281     clusterMidPoint = clusterMidPoint/len(clustersData[i])
282     clusterMidPoints[i] = clusterMidPoint
283 bar.next()
284
285 length = (len(hope))
286 clustersMidPoints = np.zeros((length,vecSize))
287 for i in range(0,length):
288     clustersMidPoints[i] = clusterMidPoints[i]
289 bar.next()
290
291 clusterDistances = pdist(clustersMidPoints)
292 bar.next()
293 bar.finish()
294 end = time.time()
295 findingClusters = end-start
296 print("finding the clusters took {0:.4} seconds.".format(
    findingClusters))
297
298
299 # metrik daten  ber  die routine sammeln
300 radiuses = np.array([])
301 densities = np.array([])
302 counts = np.array([])
303 midPointNorms = np.array([])
304 numOfNeighbors = np.array([])
305 counter = 0
306
307 for voxel in voxels:
308     radiuses = np.append(radiuses, voxel.radius)
309     densities = np.append(densities, voxel.density)
310     counts = np.append(counts, len(voxel.contData)).astype(int)
311     midPointNorms = np.append(midPointNorms, fastNorm(voxel.
    midPoint))
312     numOfNeighbors = np.append(numOfNeighbors, len(voxel.neighbors)
    ).astype(int)
313     if len(voxel.contData) == 1:
314         counter += 1
```

```
315
316 # statistiken  ber  den prozess ausgeben
317 print("")
318 endEnd = time.time()
319 fullTime = endEnd-startStart
320 printStats(voxels, radiuses, counts, counter, densities,
        numElements, hope, clustersData, maxCountIndex,
        clustersMidPoints, clusterDistances, fullTime)
321
322
323 # Plots starten
324 plotVoxels(vecSize, testData, restData, boundData, voxels,
        whichData)
325
326 # historamme plotten
327 plotBar(densities, "density", densitiesOne, whichData)
328 plotBar(radiuses, "radius", radiusesOne, whichData)
329 plotBar(counts, "counts", countsOne, whichData)
330 plotBar(midPointNorms, "norms", normsOne, whichData)
331 plotBar(numOfNeighbors, "neighbors", numOfNeighborsOne, whichData)
332
333
334 # output datei  f  r  menschen schreiben
335 file = open("output-{}".format(whichData), "w")
336 file.write("---------- this is the output for dataSet {} ----------
        \n".format(whichData))
337 file.write("---------- Here follow some metrics ---------- \n")
338 file.write("the searchRadius for the boundary was {}. \n".format(
        searchRadius))
339 file.write("there are {} data points of which {} belong to the
        boundary. \n".format(numElements,len(boundData)))
340 file.write("the largest element in each dimension is {} and the
        smallest is {}. \n".format(maxElement, minElement))
341 file.write("the data set has a dimension of {}. \n".format(vecSize)
        )
342 file.write('\n \n')
343
344 file.write("---------- After creating voxels ---------- \n")
345 file.write("there are {} voxels. \n".format(voxelsUptoNow))
346 file.write("{} data points have been covered \n".format(countsOne.
        sum()))
347 file.write("{0:.2f}% of the data have been covered \n".format( (
        countsOne.sum()/numElements)*100 ))
348 file.write("\n")
349
350 file.write("---------- After randomly creating new voxels
        ---------- \n")
351 file.write("creating new voxels took {0:.4} seconds. \n".format(
        newVoxels))
352 file.write("{} new voxels were created. \n".format(counterNewVoxels
        ))
353 file.write("\n \n")
354
355 file.write("---------- Statistics about the full process ----------
        \n")
356 file.write("there are {} voxels \n".format(len(voxels)))
357 file.write("max radius is {0:.4f} and min radius is {1:.4f} \n".
```

```
    format(radiuses.max(),radiuses.min()))
file.write("mean radius is {0:.2f} \n".format(radiuses.mean()))
file.write("max counts is {} and min counts is {} \n".format(counts
    .max(),counts.min()))
file.write("and {} contain only one element \n".format(counter))
file.write("mean counts is {0:.2f} \n".format(counts.mean()))
file.write("max density is {0:.4f} and min density is {1:.4f} \n".
    format(densities.max(),densities.min()))
file.write("mean density is {0:.2f} \n".format(densities.mean()))
file.write("\n")
file.write("{} data points have been covered \n".format(counts.sum
    ()))
file.write("{0:.2f}% of the data have been covered \n".format( (
    counts.sum()/numElements)*100 ))
file.write("\n")
file.write("there are {} clusters. \n".format(len(hope)))
for i in clustersData:
    numberDataPoints = len(clustersData[i])
    maxCount = len(voxels[hope[i][maxCountIndex[i]]].contData)
    file.write("there are {} data points in cluster {}. \n".format(
    numberDataPoints, i))
    file.write("its weighted midPoint is at {}. \n".format(
    clustersMidPoints[i]))
    file.write("voxel with most counts midPoint is at {}. \n".
    format(voxels[hope[i][maxCountIndex[i]]].midPoint))
    file.write("it contains {0} data points, which it {1:.2f}% of
    the cluster's data. \n".format(maxCount, (maxCount/
    numberDataPoints)*100))
    file.write("it consists of {} voxels. \n".format(len(hope[i])))
    file.write("the voxel numbers are {}. \n".format(hope[i]))
    file.write("that is {:.2f}% of data. \n".format((
    numberDataPoints/numElements)*100))
    file.write("\n")

file.write("the pair wise distance between the clusters is {}. \n".
    format(clusterDistances))
file.write("\n \n")

file.write("---------- Process/step times ---------- \n")
file.write("reading the data and generating the metrics took {0:.4f
    } seconds. \n".format(readingData))
file.write("finding the boundary took {0:.4} seconds. \n".format(
    findingBound))
file.write("creating voxels took {0:.4} seconds.\n".format(
    creatingVoxel))
file.write("creating new voxels took {0:.4} seconds.\n".format(
    newVoxels))
file.write("finding the clusters took {0:.4} seconds.\n".format(
    findingClusters))
file.write("the whole process took {:.4} seconds. \n".format(
    fullTime))
file.write("\n \n")

file.write("---------- Here follow the first voxels ---------- \n")
for i in range(0,len(voxels)):
    file.write("voxelNumber: {}, midPoint: {}, radius: {} \n".
    format(voxels[i].voxelNumber, voxels[i].midPoint, voxels[i].
```

```
    radius))
396     file.write("counts: {}, density: {}, midPointNorm: {} \n".
        format(counts[i], densities[i], midPointNorms[i]))
397     file.write("its direct neighbors are: {} \n".format(voxels[i].
        neighbors[1:]))
398     file.write('\n')
399     if len(voxels) > voxelsUptoNow:
400         if voxels[i].voxelNumber == voxelsUptoNow-1:
401             file.write("---------- Here follow the new voxels
        ---------- \n")
402
403 file.write("\n")
404 file.write("---------- A sorted list with all data per voxel per
        cluster ---------- \n")
405 for i in range(0,len(hope)):
406     file.write("---------- cluster {} ---------- \n".format(i))
407     for j in range(0,len(hope[i])):
408         file.write("voxel number {} at {} with radius {}:\n".format
        (voxels[hope[i][j]].voxelNumber, voxels[hope[i][j]].midPoint,
        voxels[hope[i][j]].radius))
409         file.write("it contains the points:\n")
410         for point in voxels[hope[i][j]].contData:
411             file.write("{}\n".format(point))
412         file.write("\n\n")
413     file.write("\n\n")
414 file.close()
415
416 # output datei f r computer schreiben
417 file = open("readout-{}".format(whichData), "w")
418 file.write("# this is the computer readable output for {}\n".format
        (whichData))
419 file.write("# it contains an unprocessed list of all voxels\n")
420 file.write("# lines beginning with 'v' are a voxel\n")
421 file.write("# the first two columns are the number and radius\n")
422 file.write("# the rest are each dimension of the mid points\n")
423 file.write("# lines beginning with '|' are the points contained in
        that voxel\n")
424 file.write("\n")
425 for voxel in voxels:
426     file.write("v {} ".format(voxel.voxelNumber))
427     file.write("{}".format(voxel.radius))
428     for x in voxel.midPoint:
429         file.write(" {}".format(x))
430     file.write("\n")
431     for point in voxel.contData:
432         file.write("|")
433         for x in point:
434             file.write(" {}".format(x))
435         file.write("\n")
436     file.write("\n")
437 file.close()
```

Listing 4: Main Program

```
1 import numpy as np
2 import random
3 import math
4 import copy
```

```
from progress.bar import ChargingBar
from scipy.spatial.distance import pdist
from scipy.spatial.distance import cdist

def fastNorm(vec):
    return math.sqrt(np.dot(vec, vec.T))

def fastDiff(vec, uec):
    return math.sqrt(np.dot((vec-uec), (vec-uec).T))

class spherus:
    def __init__(self, voxelNumber: int, midPoint: np.array, radius: float, contData = np.array([]), density = 0.0, neighbors = np.array([])):
        self.voxelNumber = voxelNumber
        self.midPoint = midPoint
        self.radius = radius
        self.contData = contData
        self.density = density
        self.neighbors = neighbors
    def moveMidPoint(self, speed: float, ankerPoint: np.array):
        self.midPoint += speed*(self.midPoint - ankerPoint)
    def growRadius(self, point: np.array, distance: float):
        self.radius = fastDiff(self.midPoint, point) - distance
    def appendData(self, dataSet: np.array, vecSize: int):
        norms = cdist([self.midPoint],dataSet).ravel() - self.radius
        indices = np.where(norms<=0)
        self.contData = np.append(self.contData, dataSet[indices])
        vertSize = np.round(len(self.contData)/vecSize,0).astype(int)
        self.contData = self.contData.reshape(vertSize,vecSize)
        self.density = len(self.contData)/self.radius
        return indices
    def findNeighbors(self, point: np.array, distance: float, number: int, searchRadius: float):
        if fastDiff(self.midPoint, point) - (self.radius+distance) <= 0:
            self.neighbors = np.append(self.neighbors, number).astype(int)
    def checkCollision(self, compareVoxel):
        if fastNorm(self.midPoint-compareVoxel.midPoint) > (compareVoxel.radius + self.radius):
            return False
        return True
    def findNextVoxel(self, compareVoxels, k = 1):
        normList = np.array([])
        for vox in compareVoxels:
            norm = fastNorm(vox.midPoint-self.midPoint) - vox.radius
            normList = np.append(normList, norm)
        closest = np.argsort(normList)
        return closest[:k]

class cubus:
    def __init__(self, voxelNumber: int, midPoint: np.array, edge: float, contData = np.array([]), boundData = np.array([]),
```

```
    borderData = np.array([]), density = 0.0, neighbors = np.array
    ([])):
52      self.voxelNumber = voxelNumber
53      self.midPoint = midPoint
54      self.edge = edge
55      self.contData = contData
56      self.boundData = boundData
57      self.borderData = borderData
58      self.density = density
59      self.neighbors = neighbors
60  def appendData(self, dataSet: np.array, vecSize: int):
61      for testPoint in dataSet:
62          counter=0
63          for dim in range(0,vecSize):
64              if (testPoint[dim] > self.midPoint[dim]-self.edge
    /2) and (testPoint[dim] < self.midPoint[dim]+self.edge/2):
65                  counter += 1
66          if counter == vecSize:
67              self.contData=np.append(self.contData, testPoint)
68              verticalSize = np.round(len(self.contData)/vecSize
    ,0).astype(int)
69              self.contData=np.reshape(self.contData,(
    verticalSize,vecSize))
70      return self.contData
71  def checkCollision(self, compareVoxel):
72      # das funktioniert nur, weil die voxel auf einem gitter
    sitzen und gleich gross sind
73      if fastNorm(self.midPoint-compareVoxel.midPoint) == self.
    edge:
74          return True
75      return False
76
77 def getMetrics(dataSet: np.array):
78      maxData = dataSet.max(axis=0) # gr   te Daten im Input
79      maxElement = np.ceil(maxData) # runden der max Werte des Inputs
80      minData = dataSet.min(axis=0) # kleinste Daten im Input
81      minElement = np.floor(minData) # runden der minmal Werte des
        Inputs
82      meanElement = dataSet.mean(axis=0) # gr   te Daten im Input
83      vecSize = np.size(dataSet, axis=1) # Gr   e des Vektors
84      numElements = np.size(dataSet, axis=0) # Anzahl an Elementen
85      return (minElement, maxElement, meanElement, vecSize,
        numElements)
86
87 def findNextNeighbor(dataPoint: np.array, dataSet: np.array, k = 1)
    :
88      normList = np.array([])
89      for element in dataSet:
90          norm = fastNorm(element-dataPoint)
91          normList = np.append(normList, norm)
92      closest = np.argsort(normList)
93      return closest[:k]
94
95 def findNextVoxel(testPoint: np.array, compareVoxels: spherus, k =
    1):
96      normList = np.array([])
97      for vox in compareVoxels:
```

```
98          norm = fastNorm(vox.midPoint-testPoint) - vox.radius
99          normList = np.append(normList, norm)
100     closest = np.argsort(normList)
101     return closest[:k]
102 
103 def closestDistance(dataPoint: np.array, dataSet: np.array):
104     norms = cdist([dataPoint],dataSet).ravel()
105     index = norms.argmin()
106     return index, norms[index]
107 
108 def findSearchRadius(dataSet: np.array):
109     searchData = np.unique(dataSet, axis=0)
110     numOfSelection = np.round(len(searchData)/4,0).astype(int)
111     randData = searchData[np.random.randint(0,len(searchData),
    numOfSelection)]
112     norms = np.array([])
113     length = len(randData)
114     print("calculating searchRadius for boundary...")
115     bar = ChargingBar(" ", max=length)
116     for point in randData:
117         distance = cdist([point],randData)
118         norm = distance[np.nonzero(distance)].min()
119         norms = np.append(norms,norm)
120         bar.next()
121     bar.finish()
122     searchRadiusMax = norms.max()
123     searchRadiusMean = norms.mean()
124     searchRadiusMin = norms.min()
125     return searchRadiusMax, searchRadiusMean, searchRadiusMin
126 
127 def findBound(dataSet: np.array, minElement: np.array, maxElement:
    np.array, trails = 1000, searchRadius = 1.0):
128     boundIndex = np.array([])
129     overShot = np.ceil((maxElement-minElement)*0.65) + searchRadius
130     print("searching for boundary...")
131     bar = ChargingBar(" ", max=trails)
132     for i in range(0,trails):
133         midPoint = np.random.randint(minElement-overShot,maxElement
    +overShot)
134         index, distance = closestDistance(midPoint, dataSet)
135         if  distance > searchRadius:
136             boundIndex = np.append(boundIndex, index)
137         bar.next()
138     bar.finish()
139     return np.unique(boundIndex.astype(int))
140 
141 def splitData(dataSet: np.array, boundIndex: np.array):
142     boundData = dataSet[boundIndex]
143     restData = copy.copy(dataSet)
144     restData = np.delete(restData,boundIndex,axis=0)
145     return boundData, restData
146 
147 def pickMidPoint(dataSet: np.array, vecSize: int, k = 18):
148     index = np.random.choice(len(dataSet),1)[0]
149     pick = dataSet[index]
150     neighborsIndex = findNextNeighbor(pick, dataSet, k)
151     neighbors = dataSet[neighborsIndex]
```

```
    allNeighbors = copy.copy(neighbors)
    norms = np.array([])
    for point in neighbors:
        norms = np.append(norms,fastNorm(pick-point))
    meanNorm = np.mean(norms)
    stepArray = np.array([])
    for point in neighbors:
        if fastNorm(pick-point)<meanNorm:
            stepArray = np.append(stepArray, point)
            vertSize = np.round(len(stepArray)/vecSize,0).astype(int)
            stepArray = stepArray.reshape(vertSize,vecSize)
    neighbors = stepArray
    return np.mean(neighbors,axis=0), neighbors, allNeighbors

def meanPoint(pick: np.array, dataSet: np.array, vecSize: int, k = 18):
    norms = cdist([pick],dataSet).ravel()
    neighborsIndex = norms.argsort()[1:k+1]
    neighbors = dataSet[neighborsIndex]
    allNeighbors = copy.copy(neighbors)
    norms = np.array([])
    for point in neighbors:
        norms = np.append(norms,fastNorm(pick-point))
    meanNorm = np.mean(norms)
    stepArray = np.array([])
    for point in neighbors:
        if fastNorm(pick-point)<meanNorm:
            stepArray = np.append(stepArray, point)
            vertSize = np.round(len(stepArray)/vecSize,0).astype(int)
            stepArray = stepArray.reshape(vertSize,vecSize)
    neighbors = stepArray
    return np.mean(neighbors,axis=0), neighbors, allNeighbors

def checkCollision(distanceVec: np.array, radius: float):
    if fastNorm(distanceVec) > radius:
        return False
    return True

def closestNumber(inputNumber, baseNumber) :
    quotient = int(inputNumber / baseNumber)
    numberOne = baseNumber * quotient

    if((inputNumber * baseNumber) > 0) :
        numberTwo = (baseNumber * (quotient + 1))
    else :
        numberTwo = (baseNumber * (quotient - 1))

    if (abs(inputNumber - numberOne) < abs(inputNumber - numberTwo)) :
        return numberOne
    else:
        return numberTwo

def creatMidPoints(dataSet: np.array, vecSize: int, stepSize: float):
```

```
    midPoints = np.array([])
    length = len(dataSet)
    print("creating midPoints...")
    bar = ChargingBar(" ", max=length)
    for point in dataSet:
        midPoint = np.zeros(vecSize)
        for dim in range(0,vecSize):
            midPoint[dim] = closestNumber(point[dim], stepSize)
        midPoints = np.append(midPoints,midPoint)
        vertSize = np.round(len(midPoints)/vecSize,0).astype(int)
        midPoints = midPoints.reshape(vertSize,vecSize)
        bar.next()
    bar.finish()
    return np.unique(midPoints, axis=0)
```

Listing 5: Code including the functions used in the main Program

```
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

def printStats(voxels, radiuses, counts, counter, densities,
    numElements, hope, clustersData, maxCountIndex,
    clustersMidPoints, clusterDistances, fullTime):
    print("there are {} voxels".format(len(voxels)))
    print("max radius is {0:.4f} and min radius is {1:.4f}".format(
    radiuses.max(),radiuses.min()))
    print("mean radius is {0:.2f}".format(radiuses.mean()))
    print("max counts is {} and min counts is {}".format(counts.max
    (),counts.min()))
    print("and {} countain only one element".format(counter))
    print("mean counts is {0:.2f}".format(counts.mean()))
    print("max density is {0:.4f} and min density is {1:.4f}".
    format(densities.max(),densities.min()))
    print("mean density is {0:.2f}".format(densities.mean()))
    print("")
    print("{} data points have been covered".format(counts.sum()))
    print("{0:.2f}% of the data have been covered".format( (counts.
    sum()/numElements)*100 ))
    print("")
    print("there are {} clusters.".format(len(hope)))
    for i in clustersData:
        numberDataPoints = len(clustersData[i])
        maxCount = len(voxels[hope[i][maxCountIndex[i]]].contData)
        print("there are {} data points in cluster {}.".format(
    numberDataPoints, i))
        print("its weighted midPoint is at {}.".format(
    clustersMidPoints[i]))
        print("voxel with most counts midPoint is at {}.".format(
    voxels[hope[i][maxCountIndex[i]]].midPoint))
        print("it contains {0} data points, which it {1:.2f}% of
    the cluster's data.".format(maxCount, (maxCount/
    numberDataPoints)*100))
        print("it consists of {} voxels.".format(len(hope[i])))
        print("the voxel numbers are {}.".format(hope[i]))
        print("that is {:.2f}% of data.".format((numberDataPoints/
    numElements)*100))
        print("")
```

```
30
31      print("the pair wise distance between the clusters is {}.".
        format(clusterDistances))
32      print("")
33      print("everything together took {:.4} seconds.".format(fullTime
        ))
34      print("")
35
36  def plotVoxels(vecSize, testData, restData, boundData, voxels,
        whichData):
37      if vecSize == 2:
38          fig, ax = plt.subplots()
39
40          plt.scatter(testData[:,0],testData[:,1], marker='.', c='
        #007700', label='covered data')
41          plt.scatter(restData[:,0],restData[:,1], marker='.', label=
        'not covered')
42          plt.scatter(boundData[:,0],boundData[:,1], marker='.',
        label='boundData')
43
44          for voxel in voxels:
45              plt.scatter(voxel.midPoint[0], voxel.midPoint[1],
        marker='.', c='#000000')
46              circ = plt.Circle((voxel.midPoint[0], voxel.midPoint
        [1]), voxel.radius, fill=False)
47              plt.annotate(voxel.voxelNumber, (voxel.midPoint[0],
        voxel.midPoint[1]), c='#000000')
48              ax.add_artist(circ)
49
50          plt.axis('equal')
51          ax.set_xlabel('X Label')
52          ax.set_ylabel('Y Label')
53          ax.legend()
54          plt.savefig("voxelstochs-{}.png".format(whichData), dpi
        =300)
55          plt.show()
56
57      if vecSize == 3:
58          fig = plt.figure()
59          ax = fig.add_subplot(111, projection='3d')
60
61          u = np.linspace(0, 2 * np.pi, 100)
62          v = np.linspace(0, np.pi, 100)
63
64          x = np.outer(np.cos(u), np.sin(v))
65          y = np.outer(np.sin(u), np.sin(v))
66          z = np.outer(np.ones(np.size(u)), np.cos(v))
67          for voxel in voxels:
68              ax.scatter(voxel.midPoint[0],voxel.midPoint[1], voxel.
        midPoint[2], marker='x', color='k')
69              ax.plot_surface(voxel.radius*x+voxel.midPoint[0], voxel
        .radius*y+voxel.midPoint[1], voxel.radius*z+voxel.midPoint[2],
        rstride=4, cstride=4, alpha=0.25, color='b')
70              ax.plot_wireframe(voxel.radius*x+voxel.midPoint[0],
        voxel.radius*y+voxel.midPoint[1], voxel.radius*z+voxel.midPoint
        [2], rstride=12, cstride=12, color='k')
71
```

```
        ax.scatter(restData[:,0],restData[:,1], restData[:,2],
    marker='o', label='not covered')
        ax.scatter(boundData[:,0],boundData[:,1], boundData[:,2],
    marker='o', label='boundData')
        ax.scatter(testData[:,0],testData[:,1], testData[:,2],
    marker='.', color='#777777', label='covered data')

        ax.set_xlabel('X Label')
        ax.set_ylabel('Y Label')
        ax.set_zlabel('Z Label')
        plt.savefig("voxelstochs-{}.png".format(whichData), dpi
    =300)
        plt.show()

def plotBar(dataSet: np.array, ylabel: str, dataSetOne = np.array
    ([]), prefix = ''):
    fig, ax = plt.subplots()
    plt.title("plot for {}".format(ylabel))
    ax.set_xlabel('voxel number')
    ax.set_ylabel(ylabel)
    x = np.arange(len(dataSet))
    plt.bar(x, dataSet, label='{} after all steps'.format(ylabel))
    if len(dataSetOne) > 0:
        x1 = np.arange(len(dataSetOne))
        plt.bar(x1, dataSetOne, label='{} before all steps'.format(
    ylabel))
    ax.legend()
    plt.savefig("histogram-for-{}-{}.png".format(ylabel, prefix),
    dpi=300)
```

Listing 6: Extra Functions

5.3 Systematical approach

```
import numpy as np
import matplotlib.pyplot as plt
import math
import copy
import itertools
import time
import random
from tqdm import tqdm
from scipy.spatial import distance
from scipy.spatial.distance import pdist
from scipy.spatial.distance import cdist
from Voxmidhelp import *



#testData = genData(18000)



fileDD = "Traindata/Beam_dd_cluster_sim.txt"
dataDD = np.genfromtxt(fileDD, delimiter=' ')
filePI = "Traindata/Beam_pi_cluster_sim.txt"
dataPI = np.genfromtxt(filePI, delimiter=' ')
```

```
fileT = "Traindata/Beam_T_cluster_sim.txt"
dataT = np.genfromtxt(fileT, delimiter=' ')
#fileBG = "Traindata/Beam_BG_cluster_sim.txt"
#dataBG = np.genfromtxt(fileBG, delimiter=' ')
testData = np.vstack((dataT, dataPI))
#np.random.shuffle(testData)


print(len(testData))

minElement, maxElement, vecSize, numElements = getMetrics(testData)

stepSize=15

print("Creating anchorPoints")
anchorPoints= createCornerPoints2(stepSize,testData,vecSize)
datasincubes=[]
middlePoints=[]
cubes = {}
emptycubes={}
anchorPointsLen=len(anchorPoints)



print("Creating Cubes ")
for i in tqdm(range(0,anchorPointsLen)):
    cubes[i]=Cubus(i,stepSize,(anchorPoints[i]))
    cubes[i].cubusData(testData,vecSize)
    if len(cubes[i].cubeData) == 0:
        del cubes[i]
        emptycubes[i]=Cubus(i,stepSize,(anchorPoints[i]))

for i in cubes:
    datasincubes=np.append(datasincubes, cubes[i].cubeData)


datasincubes=np.reshape(datasincubes, (int(len(datasincubes)/
    vecSize), vecSize))



fig, ax = plt.subplots()
plt.scatter(testData[:,0],testData[:,1], marker='o', label='data
    points', color='b')
plt.scatter(datasincubes[:,0],datasincubes[:,1], marker='o', label=
    'datas in voxels', color='g')




for i in cubes:
    smallrects=plt.Rectangle((cubes[i].middlePoint-cubes[i].Edge/2)
    , cubes[i].Edge, cubes[i].Edge, fill=False)
    ax.add_patch(smallrects)

plt.axis('equal')

```

```
77 ax.legend()
78 plt.savefig("density.png", dpi=300)
79
80
81 densities = np.array([])
82 counts = np.array([])
83 midPointNorms=np.array([])
84 counter = 0
85 for index in tqdm(cubes):
86     densities = np.append(densities, len(cubes[index].cubeData)/
       cubes[index].Edge)
87     counts = np.append(counts, len(cubes[index].cubeData)).astype(
       int)
88     midPointNorms = np.append(midPointNorms, fastNorm(cubes[index].
       middlePoint))
89     if len(cubes[index].cubeData) == 1:
90         counter += 1
91
92
93
94 print("Daten in W rfeln: ",  len(datasincubes), "\n")
95 print("Prozentualer Anteil von Daten in W rfeln: ", (len(
       datasincubes)/len(testData)*100), "%",  "\n")
96 print("Maximale Counts: {} Minimale Counts {}".format(counts.max(),
       counts.min()))
97 print("Mittlere Counts {0:.2f}".format(counts.mean()))
98 print("Anzahl an Voxel: {}".format(len(cubes)))
99 print("Leere Voxel {}".format(len(emptycubes)))
100 print("{} Voxel enthalten nur ein Element".format(counter))
101 print("Maximale Dichte {0:.3f} Minimale Dichte {1:.3f}".format(
       densities.max(),densities.min()))
102 print("Mittlere Dichte {0:.2f}".format(densities.mean()))
103
104
105 clustering(cubes, vecSize, testData, numElements)
106
107
108 for i in tqdm(cubes):
109     neighbors = np.array([cubes[i].number])
110     for dim in range(0,vecSize):
111         cubes[i].middlePoint[dim]+=cubes[i].Edge
112         for j in cubes:
113             if i != j:
114                 if all(cubes[i].middlePoint==cubes[j].middlePoint):
115                     neighbors = np.append(neighbors, cubes[j].
       number).astype(int)
116                     cubes[i].neighbors=np.append(cubes[i].neighbors
       , cubes[j].number)
117         cubes[i].middlePoint[dim]-=2*cubes[i].Edge
118         for j in cubes:
119             if i != j:
120                 if all(cubes[i].middlePoint==cubes[j].middlePoint):
121                     neighbors = np.append(neighbors, cubes[j].
       number).astype(int)
122                     cubes[i].neighbors=np.append(cubes[i].neighbors
       , cubes[j].number)
123         cubes[i].middlePoint[dim]+=cubes[i].Edge
```

```
124
125 countsun=np.unique(counts)
126 countnrs=np.zeros(len(countsun), dtype=int)
127 neighs=np.arange(2*vecSize+1)
128 neighcounts=np.zeros(2*vecSize+1, dtype=int)
129
130 for index in cubes:
131     for i in range(0,2*vecSize+1):
132         if len(cubes[index].neighbors)==i:
133             neighcounts[i]+=1
134
135
136
137 for i in  range(0,len(countsun)):
138     for index in cubes:
139         if len(cubes[index].cubeData)==countsun[i]:
140             countnrs[i]+=1
141
142 plotBar(counts, "counts")
143 plotBar2(countsun, countnrs, "Number of counts")
144 plotBar2(neighs, neighcounts, "Number of neighbors")
145 plotBar(midPointNorms, "norms")
146
147
148
149 plt.show()
```

Listing 7: Main Program

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import math
4  import copy
5  import itertools as itt
6  import itertools
7  import time
8  import random
9  from tqdm import tqdm
10 from scipy.spatial import distance
11 from scipy.spatial.distance import pdist
12 from scipy.spatial.distance import cdist
13
14
15 class Cubus :
16     def __init__(self, number: int, Edge: float, middlePoint: np.
       array, cubeData=np.array([]), neighbors=np.array([])):
17         self.number = number
18         self.Edge = Edge
19         self.middlePoint = middlePoint
20         self.cubeData=cubeData
21         self.neighbors=neighbors
22
23     def cubusData(self, data: np.array, vecSize: int):
24         for Element in data:
25             counter=0
26             for dim in range(0,vecSize):
27                 if (self.middlePoint[dim]-self.Edge/2 < Element[dim
       ]) and (Element[dim] < self.middlePoint[dim]+self.Edge/2):
```

```
                    counter += 1
            if counter == vecSize:
                self.cubeData=np.append(self.cubeData, Element)
        verticalSize = np.round(len(self.cubeData)/vecSize,0).
    astype(int)
        self.cubeData=np.reshape(self.cubeData,(verticalSize,
    vecSize))
        return self.cubeData

def genData(number):
    n = np.round(number/3,0).astype(int)
    a = np.random.multivariate_normal([5, -5], [[2, 1], [1, 1.5]],
    size=n)
    b = np.random.multivariate_normal([5, 5], [[2, 1], [1, 1.5]],
    size=n)
    c = np.random.multivariate_normal([-5, -5], [[2, 1], [1, 1.5]],
     size=n)
    z = np.concatenate((a, b, c))
    return z


def getMetrics(data: np.array):
    maxData = data.max(axis=0) # gr   te Daten im Input
    maxElement = np.ceil(maxData).astype(int) # runden der max
    Werte des Inputs
    minData = data.min(axis=0) # kleinste Daten im Input
    minElement = np.floor(minData).astype(int) # runden der minmal
    Werte des Inputs
    vecSize = np.size(data, axis=1) # Gr   e des Vektors
    numElements = np.size(data, axis=0) # Anzahl an Elementen
    return (minElement, maxElement, vecSize, numElements)


def createCornerPoints2(stepSize: float, data: np.array, vecSize:
    int):
    midPoints=[]
    for Elements in tqdm(data):
        for dim in range(0,vecSize):
            midPoints=np.append(midPoints,closestNumber(Elements[
    dim],stepSize))
    midPoints=np.reshape(midPoints,(int(len(midPoints)/vecSize),
    vecSize))
    midPoints=np.unique(midPoints, axis=0)
    return midPoints


def closestNumber(n, m) :
    # Find the quotient
    q = int(n / m)

    # 1st possible closest number
    n1 = m * q

    # 2nd possible closest number
    if((n * m) > 0) :
        n2 = (m * (q + 1))
    else :
```

```
        n2 = (m * (q - 1))

    # if true, then n1 is the required closest number
    if (abs(n - n1) < abs(n - n2)) :
        return n1

    # else n2 is the required closest number
    return n2

def fastNorm(vec):
    return math.sqrt(np.dot(vec, vec.T))



def plotBar(dataSet: np.array, ylabel: str, dataSetOne = np.array([]), prefix = ''):
    fig, ax = plt.subplots()
    plt.title("plot for {}".format(ylabel))
    ax.set_xlabel('voxel number')
    ax.set_ylabel(ylabel)
    x = np.arange(len(dataSet))
    plt.bar(x, dataSet, label='{}'.format(ylabel))
    ax.legend()
    plt.savefig("histogram-for-{}-{}.png".format(ylabel, prefix), dpi=300)

def plotBar2(x: np.array, dataSet: np.array, ylabel: str, dataSetOne = np.array([]), prefix = ''):
    fig, ax = plt.subplots()
    plt.title("plot for {}".format(ylabel))
    ax.set_xlabel('counts')
    ax.set_ylabel("# Voxel")
    plt.bar(x, dataSet, label='{}'.format(ylabel))
    ax.legend()
    plt.savefig("histogram-for-{}-{}.png".format(ylabel, prefix), dpi=300)




def clustering(cubes: dict, vecSize: int, testData: np.array, numElements: int):
    start = time.time()
    print("looking for neighbors...")
    clusters = []
    for i in tqdm(cubes):
        neighbors = np.array([cubes[i].number])
        for dim in range(0,vecSize):
            cubes[i].middlePoint[dim]+=cubes[i].Edge
            for j in cubes:
                if i != j:
                    if all(cubes[i].middlePoint==cubes[j].middlePoint):
                        neighbors = np.append(neighbors, cubes[j].number).astype(int)
                        cubes[i].neighbors=np.append(cubes[i].neighbors, cubes[j].number)
```

```
            cubes[i].middlePoint[dim]-=2*cubes[i].Edge
            for j in cubes:
                if i != j:
                    if all(cubes[i].middlePoint==cubes[j].
middlePoint):
                        neighbors = np.append(neighbors, cubes[j].
number).astype(int)
                        cubes[i].neighbors=np.append(cubes[i].
neighbors, cubes[j].number)
            cubes[i].middlePoint[dim]+=cubes[i].Edge
        #print(neighbors)
        if(len(neighbors) > 1) or (len(cubes[i].cubeData)/len(
testData) > 0.001):
            clusters.append(neighbors)
    clusters = np.array(clusters)
    tuples = list(itertools.permutations(range(len(clusters)), 2))
    print("Appending neighborlists...")
    for i,j in tqdm(tuples):
        if len(set(clusters[i]).intersection(clusters[j])) != 0:
            clusters[i] = np.append(clusters[i], clusters[j])
            clusters[i] = np.unique(clusters[i])
            clusters[i] = np.sort(clusters[i], kind='mergesort')
    print("Deleting doubles...")
    indices = np.array([])
    for i,j in tqdm(tuples):
        if (len(set(clusters[i]).intersection(clusters[j])) != 0)
and (i not in indices):
            if len(clusters[i]) >= len(clusters[j]):
                indices = np.append(indices, j).astype(int)
            if len(clusters[j]) > len(clusters[i]):
                indices = np.append(indices, i).astype(int)
    indices = np.unique(indices)
    hope = np.delete(clusters, indices)

    clustersData = {}
    for i in range(0,len(hope)):
        iClusterData = np.array([])
        for j in range(0,len(hope[i])):
            iClusterData = np.append(iClusterData, cubes[hope[i][j
]].cubeData)
            vertSize = np.round(len(iClusterData)/vecSize,0).astype
(int)
            iClusterData = iClusterData.reshape(vertSize,vecSize)
        clustersData[i] = iClusterData

    end = time.time()
    print("there are {} clusters.".format(len(hope)))


    for i in clustersData:
        numberDataPoints = len(clustersData[i])
        print("there are {} data points in cluster {}.".format(
numberDataPoints, i))
        print("it consists of {} voxels.".format(len(hope[i])))
        print("that is {:.2f}% of data.".format((numberDataPoints/
numElements)*100))
```

```
172     print("finding the clusters took {0:.4} seconds.".format(end-
        start))
```

Listing 8: Code including the functions used in the main Program

References

[1] Katharina Dort. Search for highly ionizing particles withthe pixel detector in the belle ii experiment. Master's thesis, II. Physics Institute Faculty 07 Justus-Liebig-Universität Gießen, May 2019.

[2] Stephanie Käs. Multiparameter analysis of the belle ii pixeldetector's data. Master's thesis, II. Physics Institute Faculty 07 Justus-Liebig-Universität Gießen, Juli 2019.

[3] Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. On clustering validation techniques. *Journal of intelligent information systems*, 17(2-3):107–145, 2001.

[4] Desy Group. Desy - belle and belle ii - superkekb and belle ii.

[5] Max-Planck-Institute for physics. Belle ii: A pixel vertex detector for an updated accelerator.

[6] Saima Bano and MNA Khan. A survey of data clustering methods. *International Journal of Advanced Science and Technology*, 113:133–142, 2018.

[7] Belle II. Belle ii - detector.

[8] Andreas Moll. Comprehensive study of the background for the pixel vertex detector at belle ii. Juli 2015.

[9] Thomas Geßler. *Development of FPGA-Based Algorithmsfor the Data Acquisitionof the Belle II Pixel Detector*. PhD thesis, II. Physics Institute-Faculty 07Justus-Liebig-Universität Gießen, 2015.

[10] Tetsuo Abe, I Adachi, K Adamczyk, S Ahn, H Aihara, K Akai, M Aloi, L Andricek, K Aoki, Y Arai, et al. Belle ii technical design report. *arXiv preprint arXiv:1011.0352*, 2010.

[11] Alfred Ultsch. U*matrix: a tool to visualize clusters in high dimensional data. 01 2003.

[12] Lennart Ljung and Jonas Sjöberg. Overtraining, regularization and searching for minimum in neural networks, 1991.

[13] David L Donoho et al. High-dimensional data analysis: The curses and blessings of dimensionality. *AMS math challenges lecture*, 1(2000):32, 2000.