Clustering Algorithms in the Electronic Calorimeter at LDMX: Achieving Electron Counting

Ella Viirola
Summer Project 2024
LDMX at Lund University
Supervisors: Lene-Kristian Bryngemark, Hannah Herde, Ruth Pöttgen, Einar Elén

Contents

1 Introduction

I worked as a project assistant at the Lund branch of the LDMX experiment during the summer of 2024. My main goal was to lay some groundwork for the multi-electron paper by seeing if it was possible to achieve electron counting in the electronic calorimeter (ECal). Electron counting is important for being able to analyze the results when multiple electrons enter the detector at the same time, and as the tracking scintillator can undercount the number of electrons, the ECal electron counting is needed for verification. Electron counting could be achieved by forming groups or *clusters* of similar reconstructed hits (rechits) in the ECal; these would then represent the initial electrons. I analyzed different clustering algorithms and tried to find the best one for this purpose.

For help in analyzing the results of the clustering, I also set up a visualisation for rechits and clusters in the ECal using Phoenix.

2 Visualisation

As a tool for being able to assess how clustering performs, I implemented a visualisation tool that uses Phoenix, called LDMX-VIS, as well as a event data-to-JSON converter, called VisGenerator.

Most of the visualisation is under the `ella-dev` GitHub branch, with some updates in the `ella-dev-clustering` branch. (With reservation for change after pull requests go through.)

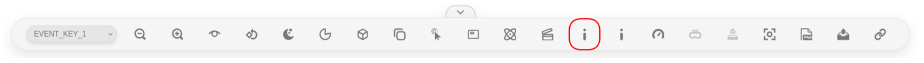
2.1 LDMX-VIS: Phoenix implementation

LDMX-VIS is the LDMX implementation of Phoenix, an event display framework. It is located under the `EventDisplay/ldmx-vis` folder.

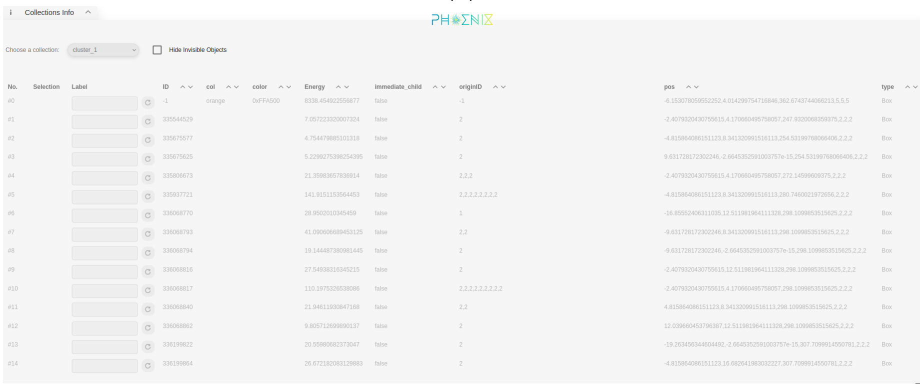
2.1.1 How to use

A guide for installing the necessary packages and running LDMX-VIS is in the `README.md` file in the `EventDisplay/ldmx-vis` folder. After the first installation steps, subsequent runs can be started with just `yarn start`, and the application will run at `http://localhost:4200`.

The application will start with a model of the detector and some default two-electron events. In the Phoenix menu at the top right, there are some visibility settings; you can set the colour and visibility of different event collections and detector parts, as well as the opacity of the latter. In the bottom menu, you can view information about the event data in the *Event data collections info* panel, see Figure 1. You can also import new event data or geometries (detector parts), see Figure 2. Note that the LDMX detector files are in GDML, which is not one of the allowed formats, so conversion is needed, see section 2.1.2. Event data also needs to be converted from ROOT files to e.g. JSON, for which the VisGenerator was built, see section 2.2.



(a)



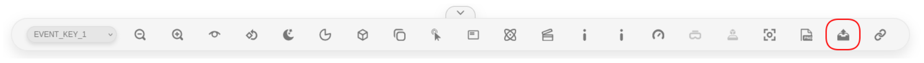
Choose a collection: cluster_1

Hide Invisible Objects

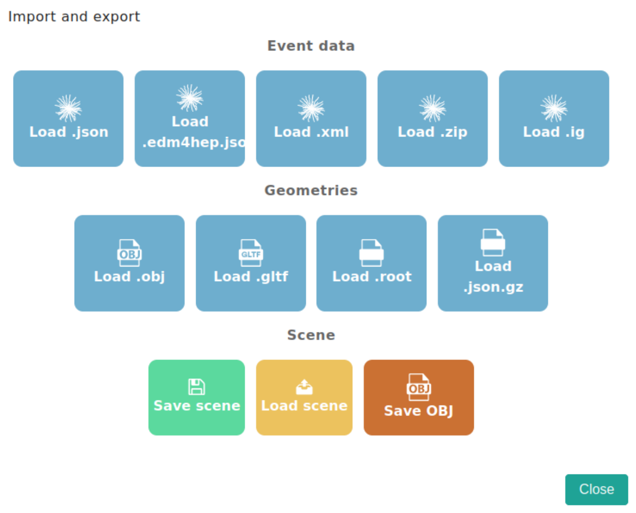
No.	Selection	Label	ID	col	color	Energy	immediate_child	originID	pos	type
#0			-1	orange	0xFFA500	8338.454922556877	false	-1	-6.153078059552252,4.014299754716846,362.6743744066213,5,5,5	Box
#1			335544529			7.057223320007324	false	2	-2.4079320430755615,4.170660495758057,247.9320068359375,2,2,2	Box
#2			335675577			4.754479885101318	false	2	-4.815864086151123,8.341320991516113,254.53199768066406,2,2,2	Box
#3			335675625			5.2299275398254395	false	2	9.631728172302246,-2.6645352591003757e-15,254.53199768066406,2,2,2	Box
#4			335806673			21.35983657836914	false	2,2,2	-2.4079320430755615,4.170660495758057,272.14599609375,2,2,2	Box
#5			335937721			141.9151153564453	false	2,2,2,2,2,2,2,2	-4.815864086151123,8.341320991516113,280.7460021972656,2,2,2	Box
#6			336068770			28.9502010345459	false	1	-16.85552406311035,12.511981964111328,298.1099853515625,2,2,2	Box
#7			336068793			41.090606689453125	false	2,2	-9.631728172302246,8.341320991516113,298.1099853515625,2,2,2	Box
#8			336068794			19.144487380981445	false	2	-9.631728172302246,-2.6645352591003757e-15,298.1099853515625,2,2,2	Box
#9			336068816			27.54938316345215	false	2	-2.4079320430755615,12.511981964111328,298.1099853515625,2,2,2	Box
#10			336068817			110.1975326538086	false	2,2,2,2,2,2,2,2,2	-2.4079320430755615,4.170660495758057,298.1099853515625,2,2,2	Box
#11			336068840			21.94611930847168	false	2,2	4.815864086151123,8.341320991516113,298.1099853515625,2,2,2	Box
#12			336068862			9.805712699890137	false	2	12.039660453796387,12.511981964111328,298.1099853515625,2,2,2	Box
#13			336199822			20.55980682373047	false	2	-19.263456344604492,-2.6645352591003757e-15,307.7099914550781,2,2,2	Box
#14			336199864			26.672182083129883	false	2	-4.815864086151123,16.682641983032227,307.7099914550781,2,2,2	Box

(b)

Figure 1: a) Where to click in the menu to get b) the event data collections info panel.



(a)



(b)

Figure 2: a) Where to click in the menu to get b) the import and export panel.

2.1.2 Build process

Note: this is a brief description of how LDMX-VIS was built; LDMX-VIS is already functional and runnable according to the instructions in the previous section.

When building the application, I created a new Phoenix angular application in `EventDisplay/ldmx-vis` using the guide in the Phoenix GitHub [6]. However, the application did not work out of the box; I had a lot of errors with the `three` package, and had to compare the code with a Phoenix application built from the Phoenix GitHub, which worked. What fixed it was to copy the packages from `package.json`, to add `.yarnrc.yml`, to change the "assets"-parameter to the actual asset folder in `angular.json`, and most importantly, to change the builder in `angular.json` to `"@angular-devkit/build-angular:browser"`. I also copied the build options from the `angular.json` file of Phoenix and copied the necessary files, namely `polyfills.ts`, `zone-flags.ts`, and `tsconfig.app.json`.

The detector was to be uploaded as a GTFL file. There is a guide in the Phoenix GitHub for converting GDML detector files, which LDMX has under the `Detector` directory, to GTFL [4], but I used a previously compiled GTFL file instead, taken from (I think) the `ldmx-event-display` project [3]. This was a previous iteration at making an LDMX Phoenix implementation. I tried building this project before starting with my own implementation, but similarly to the out-of-the-box Phoenix application, it did not work either.

Event info can be uploaded through several different file formats. Uploading the raw root files produced though `ldmx fire` does not work, however, so I chose to create a new analyzer to produce JSON files with the data instead, see section 2.2.

2.2 VisGenerator

The VisGenerator analyzer, located in the `DQM` directory, writes event data into JSON files readable by Phoenix applications. Currently it produces ECal-specific JSON visualisation files using the `nhlohmann::json` package. It writes files containing:

- **Rechits**: hit ID, hit position, total energy. IF ground truth is available (`includeGroundTruth = True`): which initial electrons have contributed to this hit OR which particles have contributed to this hit if originID is not available (see section 3.1 for information about origin ID), percentage of energy coming from each electron (AND percentage of energy coming from unknown source, if originID is not available), if the hit is an immediate child of the initial electron(s)
- **Clusters**: cluster ID, centroid position, total energy, which hits are included in the cluster (will be colored in same color)
- **Ground truth tracks** (only available with ground truth): linear extrapolation from initial electron production vertex to its endpoint

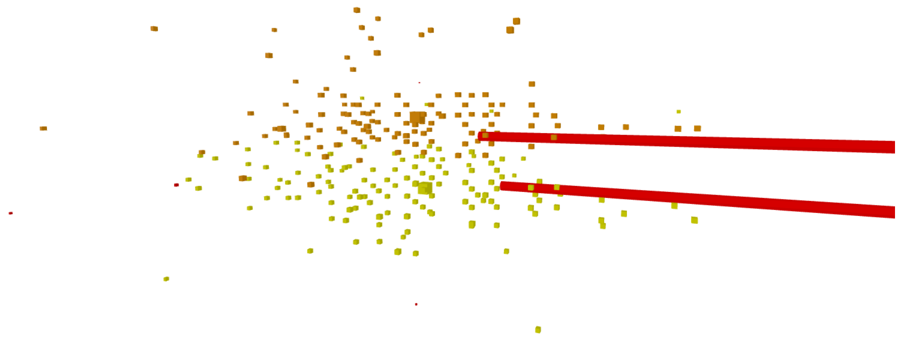


Figure 3: Example of a two-electron event with two clusters, produced by the CLUE algorithm.

All the information can be viewed in the event data collections info panel, see Figure 1. An example of an event written by VisGenerator is shown in Figure 3. Hits belonging to the same cluster have the same color; here cluster 1 is in orange, and cluster 2 is in yellow. The slightly larger boxes are the cluster centroids. The few smaller hits in red are hits that do not belong to any cluster. The two red tracks are the ground truth tracks.

There are boolean options for including only the information that you want to see in the JSON file; see `dqm.py` for all the options. Some booleans require other information; e.g. if `includeGroundTruth` is `True`, `nbrOfElectrons` needs to be provided. There are also options for only including certain events, or excluding certain events.

Additionally, VisGenerator can produce two additional files, `truth.json` and `layers.json`. `truth.json` presents rechits color-coded after which initial electron they come from (with mixed hits as a separate collection), and `layers.json` presents rechits color-coded after which ECal layer they are in. For `truth.json`, ground truth and origin ID information must be available. For `layers.json`, this is not needed.

Note that JSON files produced get quite large in size; a file with 1000 two-electron events is over 100 MB in size. The previously mentioned include/exclude options can be used to narrow down the size.

2.2.1 Expansion

New types of data can be added to VisGenerator. The types of objects allowed by Phoenix are presented in the Event Data Format page in the Phoenix developer's guide [5]. Many of the object types available are designed for radial detectors rather than solid target detectors like LDMX; I have relied heavily on the `Box` type hit, which I would recommend over the `Point` type hit, whose size cannot be modified.

The JSON files are structured as follows:

```
{
    "EVENT_KEY_NBR": {
        "Object type": {
            "collection_name": [
                {
                    "attrib1": "hi",
                    "attrib2": 3,
                    ...
                },
                {
                    ...
                }
            ],
            "collection_name_2": [...]
        },
        "Another object type": {
            ...
        }
    },
    "EVENT_KEY_NBR+1": {...}
}
```

Here, "object type" refers to the listed available object types in the Event Data Format page, e.g. Hits or Tracks.

The entire structure of the JSON file is stored in a `nlohmann::json` object with name `j` while events are processed. A `nlohmann::json` object functions very similarly to a Map. For example, the JSON object `Hits` is accessed with `j[eKey]["Hits"]`, where eKey is the event key.

To add new data, you can either create a new collection under an already existing object type (Hits or Tracks) by calling e.g. `j[eKey]["Hits"]["collection name"] = json::array();`, or add a new object type by calling `j[eKey]["Object type"] = json::object();` (and then create a new collection within that object type). As mentioned, only some object types are allowed; refer to the Event Data Format guide to see these. Collection names have no restrictions: as an example, under the object type "Hits" I have several cluster collections, each with name "cluster_i", where i is the index of the cluster, that contain the cluster centroid as well as all hits in the cluster.

When adding new objects to a collection, I found it easiest to work with a temporary object by calling `json temp = json::object();` and then adding attributes by calling `temp["attribute name"] = 1` and so on. When an object has all desired attributes, you can call `j[eKey]["object type"]["collection name"].push_back(temp)` to add it to the collection.

For each event data format there are some required attributes that are listed in the guide, but it is possible to add custom attributes that show up as information in the *Event data collections info* panel, see Figure 1. For example, I have included the energy of both clusters and rechits. These custom attributes only show up in this panel and nowhere else in Phoenix.

The attributes for a collection are modelled after the first object in the collection. In the cluster collections, for example, the first object is the cluster centroid, that does not have an ID or an originID, but these are initialized to -1 anyway, as the following rechits have these attributes. Similarly, *all* the objects in a collection will be set to the color of the *first* object. Any subsequent colors will be ignored, so it is only necessary to set the color of the first object. See Figure 1b. (Note that the col-attribute is a custom parameter that was set so that it would be easier to decode the hex.)

Finally, I'd recommend adding a boolean option in VisGenerator for including your type of objects or not, as I have done with ECal clusters and rechits. The JSON files get very large in size and it is good to be able to customize what to include so it is not unnecessarily bloated.

2.3 Future improvements

- The default detector file displayed when opening LDMX-VIS may be outdated.
- There is a bug in the Phoenix code that leads to the color of Hit-type objets (rechits, cluster centroids) not being read from the JSON file, so all of these will be in the same color unless manually set to another one in Phoenix. I submitted an issue about this to the Phoenix GitHub, and it has been solved, but has yet to be included in a release at the time of writing.
- The sizes of JSON files that VisGenerator produces are quite large because of the large amount of information. I am unsure if this is something that is possible to improve, but could be worth looking into.
- It would be great to include information from the TS; currently it is just ECal-specific visualisation.
- Ground truth tracks do not accurately depict the path an electron takes; it is just a line from birth to death.
- DQM may not be the appropriate directory for the VisGenerator.

3 Clustering

The goal of the clustering algorithm was to count electrons; thus the main goal was to achieve n clusters $==$ n electrons. An additional goal was to create clusters that represent an electron, to later be able to use the clusters to analyze e.g. the energy deposited by an electron. This meant that the energy in a cluster should ideally belong to one electron only (this metric was nicknamed purity, and is explained in section 3.1) and that a cluster should capture as much of the energy deposited by an electron as possible, meaning that in a simulated environment, the total number of unclustered hits should be low.

For the clustering, I tried several different algorithms. I tried to improve the already existing algorithm, briefly attempted to create a layerwise containment radius based algorithm, and implemented topoclustering from ATLAS and CLUE from CMS. The last produced the best results, reaching an average of 1.89 clusters for 2 electrons, with around 87.99% energy purity and 6.938% of the total number of hits unclustered.

An overwhelming majority of the analysis of the clustering method was done with inclusive samples of 8 GeV two-electron events.

Code related to clustering (`CLUE`, `EcalClusterAnalyzer`, modifications to `EcalClusterProducer`) is located in the `ella-dev-clustering` branch in GitHub. (With reservation for change after pull requests go through.)

Terminology

A cluster **seed** is the starting point for a cluster. E.g. when clustering rechits, the seed will be a rechit. Something is a seed if it passes some criteria, usually at least that it has high enough energy (passes a seed threshold).

A **centroid** is the central point of a cluster. It is calculated by weighting the position of all rechits with their energy.

3.1 Metrics

The main metrics used to determine the overall performance of a clustering method were, as mentioned, the number of clusters it produced per event, the energy purity of each cluster, as well as the percentage of hits that are not clustered in an event. The energy purity metric refers to the percentage of the energy in the cluster that comes from the same electron, more specifically from the electron that contributes most of the energy. E.g. if electron 1 contributes 75% of the energy in a cluster, the energy purity of the cluster is 75%. For two electrons, the minimum energy purity is 50%, but in the case of more than two electrons, it may drop below that.

For the purity metric, it is necessary to know which electron(s) a hit originates from. This information is not available by default. Simulated hits have `Contrib` elements, which stores the ID of each track that has contributed to the hit, and how much energy it has deposited. However, it is not possible to trace backwards to which origin electron the track comes from at the recon stage, as the `TrackMap` that contains track ancestry information is not stored after simulation, and is also too large to reasonably be stored. This was solved by updating the simulation stage to store the ID of the origin electron for each `Contrib`. A `int originID{-1}` attribute was added to the `Contrib` struct in `SimCaloriMeterHit`, as well as a private attribute `std::vector<int> originContribs_` to the class itself, and the `Contrib` methods in the class were updated accordingly.

In `EcalSD::ProcessHits`, when adding a new `Contrib`, the `TrackMap` is used to trace if the track ID descends from particles with ID 1-5, i.e. the origin electron(s) (an event will most likely not have more than five electrons). If this fails, the origin ID is set to the incident ID.

```
if (enableHitContribs_) {
    int contrib_i = hit.findContribIndex(track_id, pdg);
    if (compressHitContribs_ and contrib_i != -1) {
      hit.updateContrib(contrib_i, edep, time);
    } else {
      int origin{-1};
      auto map {getTrackMap()};
      auto incident {map.findIncident(track_id)};
      for (int i{1}; i < 6; ++i) {
        if (map.isDescendant(track_id, i, 100)) {
          origin = i;
          break;
        }
      }
      if (origin == -1) {
        origin = map.findIncident(track_id);
      }
      hit.addContrib(getTrackMap().findIncident(track_id), track_id, pdg, edep,
                     time, origin);
    }
} else { ... }
```

Most of the cluster analysis and the histogram creation is performed in a new analyzer called `EcalClusterAnalyzer`, which is located in the `ECal` directory. It calculates the energy purity of clusters, the amount of unclustered hits, and more, like the energy purity as a function of the distance between electrons. (The last is currently only available if there are two electrons; this should be modified.)

3.2 Abandoned algorithms

I spent some time with the ATLAS topoclustering algorithm, as well as a custom layerwise clustering method, before abandoning both for different reasons. These are described briefly below.

3.2.1 ATLAS Topoclustering

Topoclustering is an algorithm previously used by the ATLAS experiment. It was implemented following the description in the paper Topological cell clustering in the ATLAS calorimeters and its performance in LHC Run 1 [1]. To summarize, hits are clustered recurisvely by starting with seed hits whose energy pass a seed threshold, and looking at their neighbours. If the neighbours pass a growth threshold, the neighbour is added to the cluster and its neighbours looked at in the same way. If the neighbour does not pass the growth threshold but passes a cell filter threshold, it is added, but its neighbours are not examined. If a neighbour is a seed for another cluster, the two clusters are merged. The code I wrote for topoclustering is presented in Appendix A.2.

The main problem of topoclustering was that it was too allowing in its clustering. The concept of "neighbouring cells" does not exist in LDMX in the same way as in ATLAS, as LDMX has much higher granularity. I therefore used a similar distance based weight as for the old algorithm to determine if the hits were close enough to be considered "neighbours". However, it was very difficult to have a definition for neighbours that would include hits that were not immediately next to each other, without having the clusters merge every time. Some very elaborate cases would have been needed based on where the hit was located. I also considered not only comparing two hits, but also taking into account the centroid position, but came to the conclusion that the algorithm then would function almost identically to the initial algorithm, see section 3.3.

3.2.2 Layerwise clustering

I briefly experimented with a layerwise clustering solution, where the idea was to work through the rechits one ECal layer at a time, and creating centroids in each layer by clustering all rechits in an area within the containment radius, with the previous layer's centroids as middle points.

The main problem with getting this to work was where to start. The first layer of the ECal does not necessarily have just two hits for two electrons; there is often a hit with contributions from both electrons, for example. I tried merging the three first layers into one and using the old algorithm to find a starting point, but it produced poor results. There was also always the problem of closely located electrons, where the containment radii certainly would overlap.

After struggling with this for a while, I came to the conclusion I was trying to reinvent the wheel, and that it would produce better results to just try to find an existing algorithm that would work better.

3.3 Initial algorithm

There was an existing algorithm for ECal clustering in `TemplatedClusterProducer.h` in combination with `MyClusterWeight.h`.

The initial algorithm worked by first making all hits into `WorkingCluster` objects, that have a centroid position. The cluster objects were sorted by energy. Then for all clusters that had an energy over a seed threshold, the lesser energy clusters were looped over and the weight between these calculated. The weight was solely distance based; the closer together the cluster centroids were, the smaller the weight. There were separate weights for the xy-plane and z-axis, and the weights were calculated according to

$$w_{xy} = e^{(d_{xy}/r_{mol})^2} - 1 \tag{1}$$

$$w_z = e^{(d_z/d_{z_{char}})} - 1, \tag{2}$$

where d_{xy} and d_z were the Euclidian distances between the hits in the xy-plane and z-axis, respectively, r_{mol} was the "Molière radius" of the detector and $d_{z_{char}}$ the "characteristic cluster longitudinal variable" (both of these in citation marks, as they were labelled as such, but did not actually have the correct values of these). The initial values were $r_{mol} = 10$ and $d_{z_{char}} = 100$.

The smallest weight of **all** these calculated weights (i.e. one weight between one seed cluster and one lesser energy cluster) was saved. These two clusters were then merged; the position of the merged cluster centroid was the average position between the two, weighted by energy. This loop of calculating weights and merging the lowest weight clusters continued until the lowest weight was above a cutoff weight.

As $d_{z_{char}} > r_{mol}$, the initial weight was more "allowing" in the z-direction, merging hits from further away in that axis. This led to the area hits were included from being a rectangular cuboid around the centroid, which was narrow in the xy-dimension and longer in the z-dimension.

3.3.1 Main problems

The weight was calculated based on the position of the cluster centroid, which changed position every time a new cluster was merged into it. Hits that would originally have been close enough to be merged into the cluster might not have been merged because the centroid moved away.

Another problem was precisely that the position was always compared to the cluster centroid; this led to the cluster, as mentioned, only including hits in a narrow rectangular area around the centroid, which left a lot of hits near the tail and especially around the most energetic parts of the shower unclustered. While the area could be expanded, especially making it wider in the xy-plane would increase the risk for merged clusters.

Finally, I found that the algorithm was quite hard to improve. The only modification opportunity was the weight, and I found myself quite stuck on the best way to implement it. I would have needed to remake the weight calculation equations, and I felt I did not have the physics knowledge to write equations for the shape of an electromagnetic shower, where the position that we calculate from is always changing.

3.3.2 Modifications

The first modifications were to clean up the old code, as it had not been worked on for quite a while and was not even runnable. I fixed this and also filtered clusters so that empty or not energetic enough clusters were not included in the final cluster list.

I attempted to modify the weight, mainly by allowing hits from a larger radius from the centroid the larger the z-coordinate was. This was done by increasing r_{mol} with the z-coordinate of the lower energy hit; a hit deeper in the detector was allowed to have a larger xy-difference from the centroid. I tried implementing an equation for the actual containment radius according to values read from Figure 4, but this proved less effective than just hardcoding values based on if the z-coordinate was larger than certain thresholds. However, this still did not produce significant improvements. The main problem of the rigid limits is the large chunk of energy that is lost in the xy-plane around the most energetic parts of the shower, and this is very hard to include without the risk of merging, as previously mentioned.

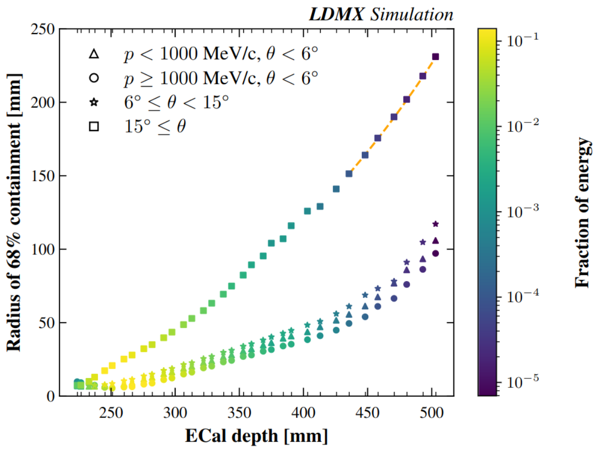


Figure 4: Containment radius in the ECal for different p and θ of the entering electron.

3.3.3 Results

The initial algorithm performs fairly well for two-electron events: for 10000 events, it had a mean number of clusters at 1.836, and a mean energy purity at 90.72%, see Figure 5 and 6, respectively. However, the percentage of hits not in a cluster is very high, at 46.32%, see Figure 7.

For three electrons, about half the events are undercounted, see Figure 8. The purity also drops to 84.3%, see Figure 9, while the percentage of clusterless hits is largely the same, see Figure 10.

The parameters used for all cases were seed threshold = 350 and cutoff = 10.

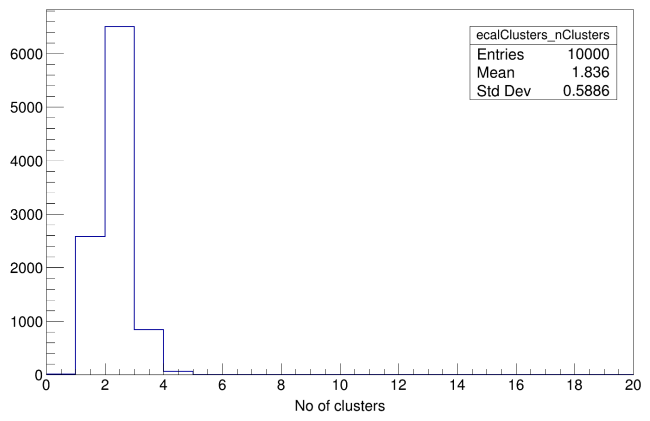


Figure 5: Number of clusters for two-electron events using the initial algorithm.

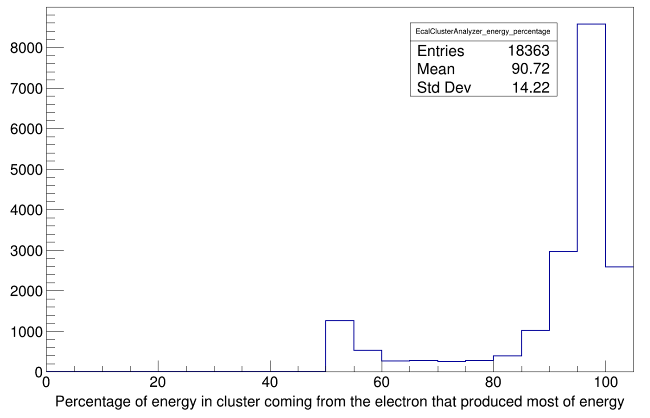


Figure 6: Energy purity for two-electron events using the initial algorithm.

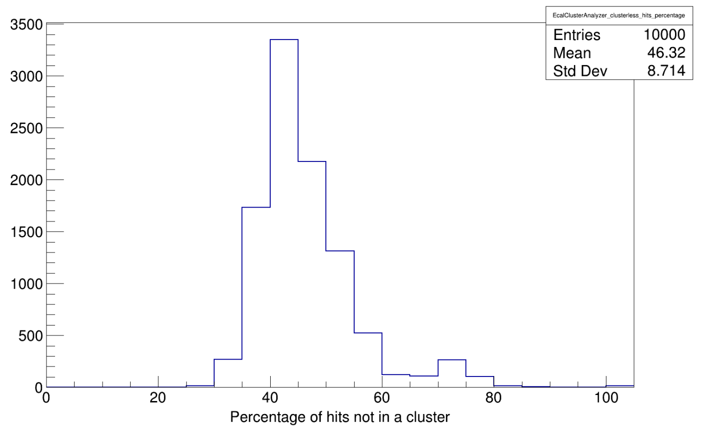


Figure 7: Percentage of hits not in a cluster for two-electron events using the initial algorithm.

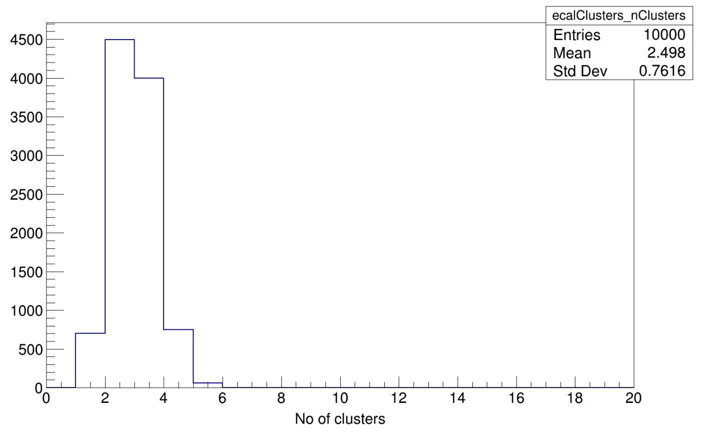


Figure 8: Number of clusters for three-electron events using the initial algorithm.

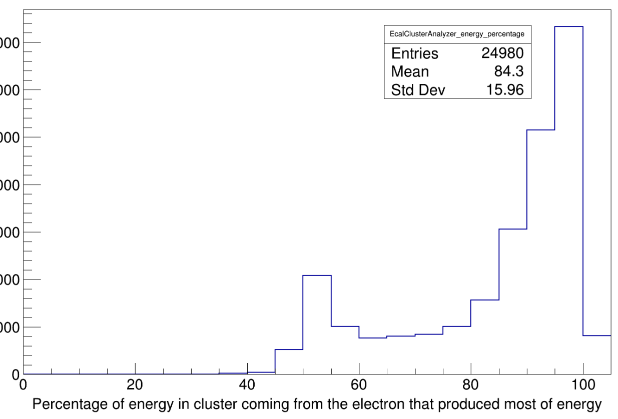


Figure 9: Energy purity for three-electron events using the initial algorithm.

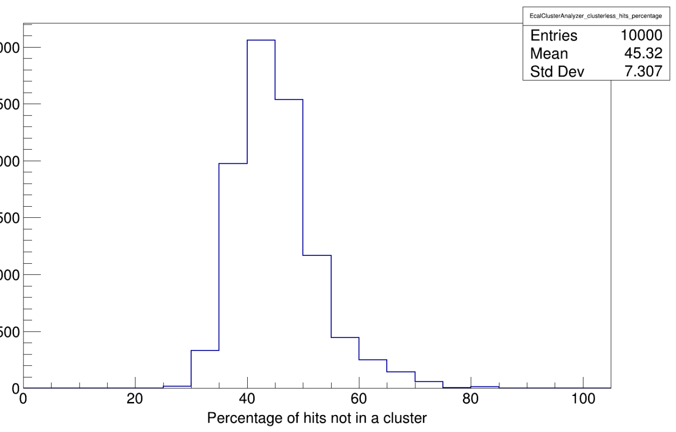


Figure 10: Percentage of hits not in a cluster for three-electron events using the initial algorithm.

3.3.4 Potential further improvements

The weight used still had potential for improvement, especially that it did not take into account the energies of the hits.

3.4 CLUE

CLUE is an algorithm used by the CMS experiment. I implemented the algorithm according to the description in the paper CLUE: A Fast Parallel Clustering Algorithm for High Granularity Calorimeters in High Energy Physics, specifically the pseudocode in Appendix A of that paper [7]. In the following sections, I will mainly explain what I did differently from the pseudocode.

The parameters necessary for CLUE are presented in Table 1. CLUE works with 2D layers. For a first version, I collapsed the z-dimension of all the hits (i.e. all hits were in the same layer), and made densities where all the hits with the same xy-coordinates were added to the same density, i.e. I did not use d_c.

Table 1: Parameters used in the CLUE algorithm.

Parameter	Description
d_c	cutoff distance in calculation of local density (size of local density)
ρ_c	minimum energy to promote density as seed/ maximum energy to demote density as outlier
δ_c	minimum separation distance for seeds
δ_o	minimum separation distance for outliers
d_m	$\max(\delta_c, \delta_o)$

3.4.1 Running the code

The CLUE algorithm is located in `Ecal/include/Ecal/CLUE.h`. It is an option for the `EcalClusterProducer`, i.e. it can be used instead of the initial algorithm when using the cluster producer. To use CLUE, set the CLUE parameter of `EcalClusterProducer` to `True`. The rest of the parameters are explained in the `Ecal/python/ecalClusters.py` file.

3.4.2 Results

The single-layer CLUE provided good results. For 10000 two-electron events, the mean number of clusters was 1.891, the mean energy purity was 87.99% and the mean percentage of hits not in a cluster was 6.938%, see Figure 11, 12, and 13. For 10000 three-electron events, the performance was slightly worse, with mean number of clusters at 2.594 and energy purity at 80.38%, but with slightly lower percentage of hits not in a cluster, at 6.361%, see Figure 14, 15, and 16.

The parameters used for both cases were $\rho_c = 550$, $\delta_c = 10$ and $\delta_o = 40$.

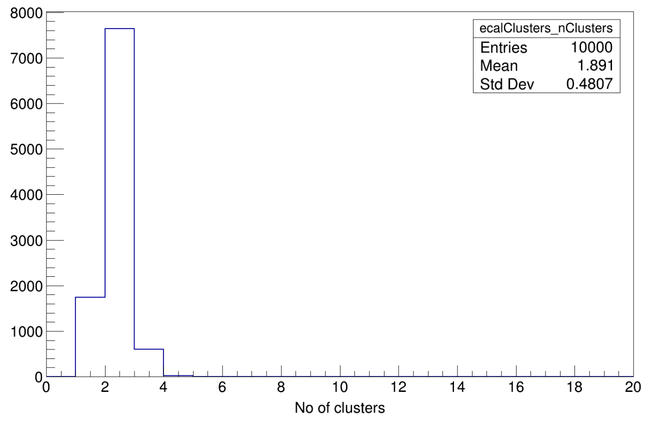


Figure 11: Number of clusters for two-electron events using CLUE.

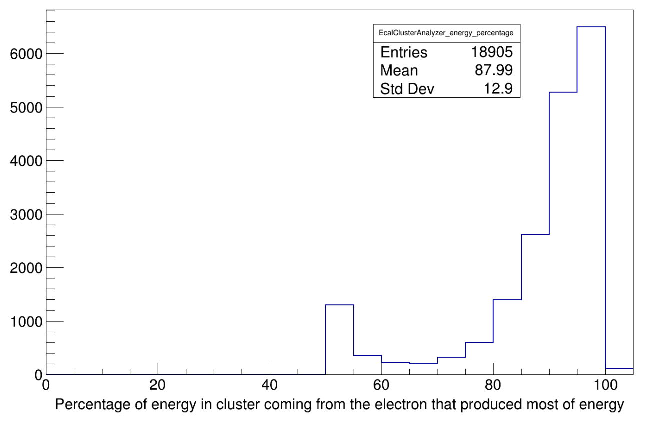


Figure 12: Energy purity for two-electron events using CLUE.

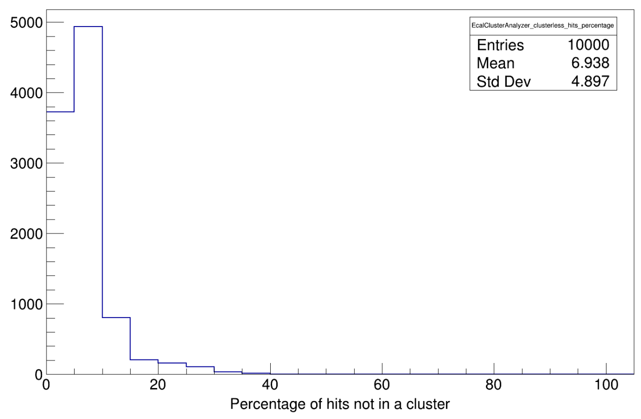


Figure 13: Percentage of hits not in a cluster for two-electron events using CLUE.

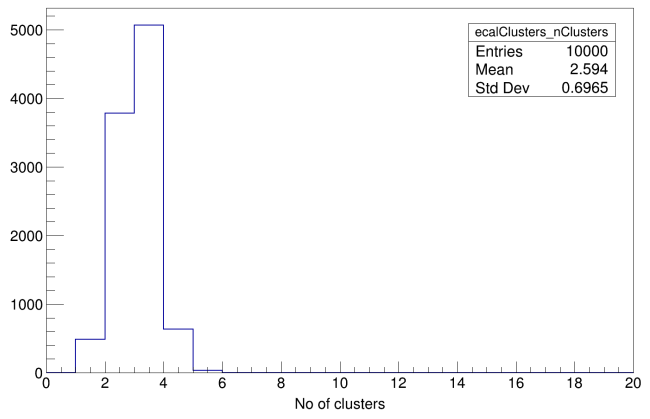


Figure 14: Number of clusters for three-electron events using CLUE.

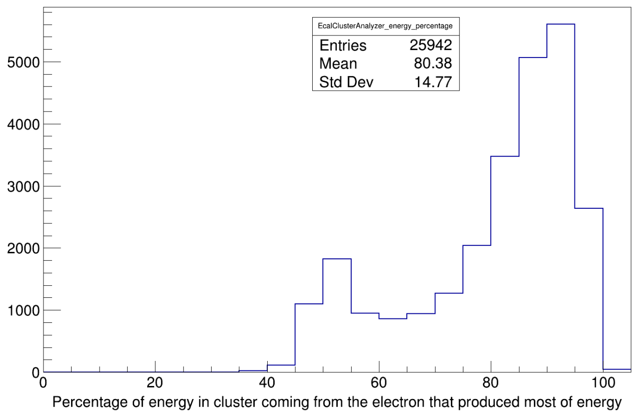


Figure 15: Energy purity for three-electron events using CLUE.

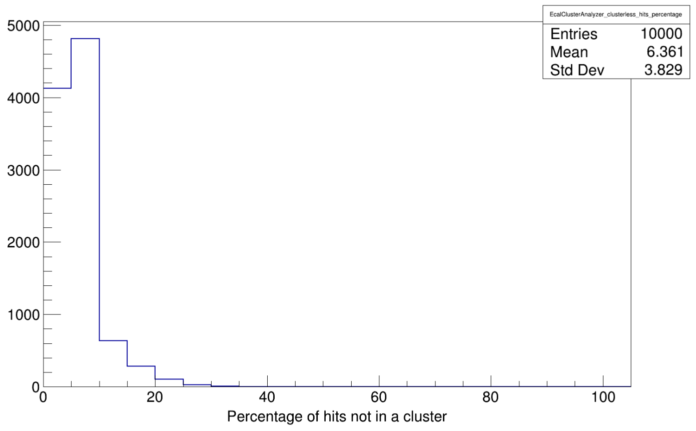


Figure 16: Percentage of hits not in a cluster for three-electron events using CLUE.

3.4.3 Reclustering

Merged clusters are still a problem in CLUE. To tackle it, I included a catch for reclustering clusters that had above 10000 MeV of energy. When marking followers of clusters, the program continuosly keeps track of the cluster energy. If it passes the max energy allowed (which is currently set to 10000 MeV, as one electron should hover around 8000 MeV; this should be double-checked), the cluster is marked as merged and the entire clustering loop is restarted with a lower δ_c. However, only densities in the merged cluster and densities that are within a certain distance of the event centroid are reclustered using the lower

δ_c; all others use the original. This is because most of the energy is centered around the event centroid, and it is usually there that the merge happens, when electrons are located close to each other. See Appendix A.1.1 for the code used.

While this reclustering method did reduce the amount of merged clusters, i.e. undercounting, it came at the cost of overcounting instead, see Figure 17 for two-electron events and Figure 18 for three-electron events. The amount of correctly counted events increases slightly in comparison to the original code (see Figure 11 and 14 for original number of clusters for two- and three-electron events, respectively), but there are now up to eight clusters in a given event. However, especially for the three-electron event, the peak of correctly counted events is more prominent. Further work on the reclustering could therefore be beneficial.

Right now, the only condition for reclustering is the cluster passing the max energy. There are potential improvements, like verifying which max energy is a suitable. It could also be helpful to look at the energy topology in the ECal, or the distance between electrons as they enter the ECal. Especially the latter seems important, as the majority of merged clusters, at least for two electrons, happen when the distance between the electrons in the xy-plane is small, see Figure 19. (As previously mentioned in section 3.1, distance analysis has not been generalized to more than two electrons and should be expanded.) However, the distance calculation I have used so far is using ECal scoring plane hits, which are only available during simulation, and thus a different method should be used for the purpose of reclustering. An option could be to look at how spread out the rechits are in the first layers – reasonably a more spread out density of hits (ignoring outliers) should mean the electrons enter further apart.

Reclustering is currently an option that can be turned on or off with the `reclustering` boolean in `ecalClusters.py`.

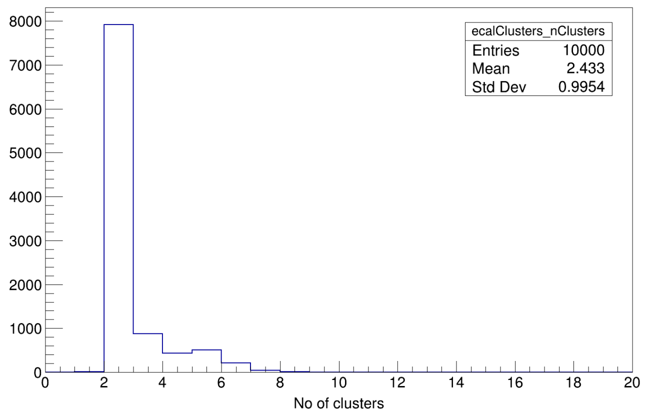


Figure 17: Number of clusters for two-electron events using CLUE and reclustering.

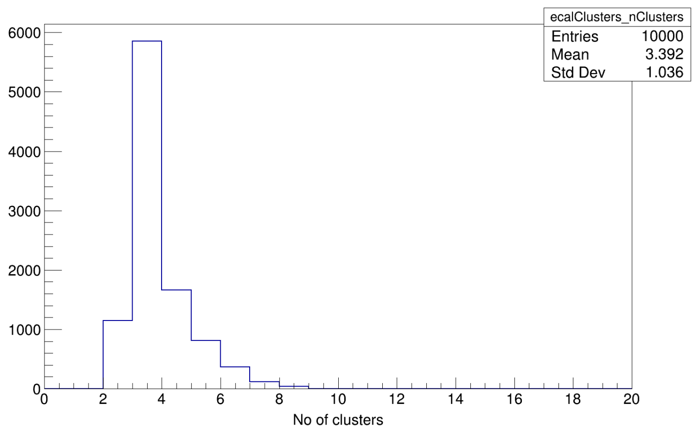


Figure 18: Number of clusters for three-electron events using CLUE and reclustering.

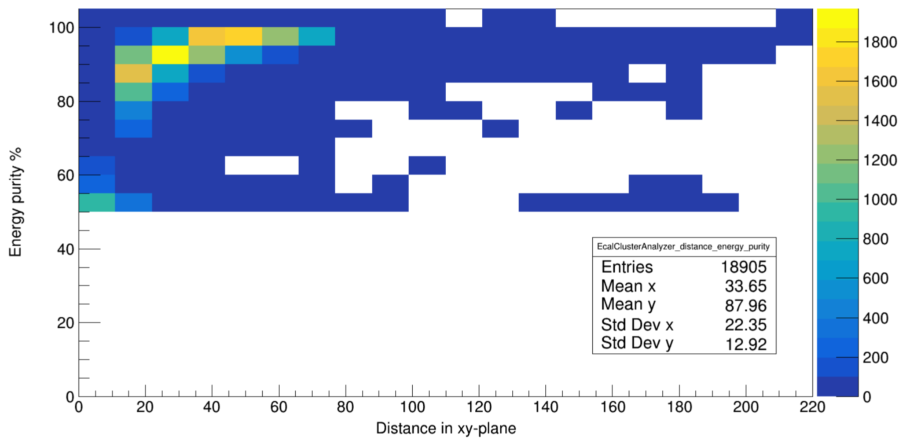
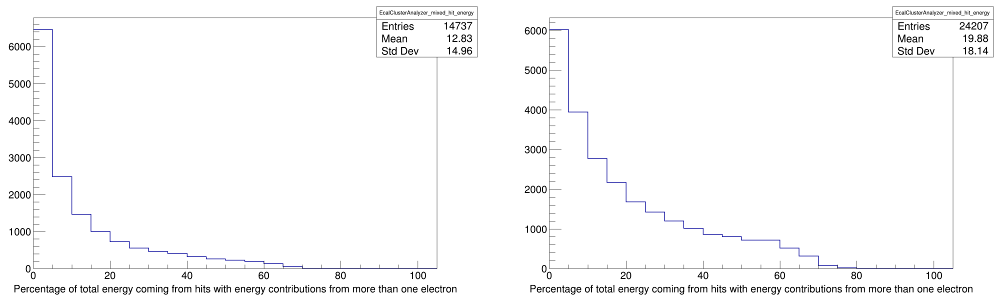


Figure 19: Energy purity as a function of distance between electrons in the ECal scoring plane, for a two-electron event using CLUE without reclustering.

3.4.4 Mixed hits

Many hits have contributions from several electrons, and including these in a cluster will automatically lead to some impurity. The more electrons in an event, the larger the percentage of a cluster's energy that comes from mixed hits; for two electrons, the mean is 12.83%, while for three electrons, the mean is 19.88%, see Figure 20.



(a) **(b)**

Figure 20: Percentage of energy of a cluster coming from mixed hits for the a) two-electron case and b) three-electron case.

I tried implementing a method that would allow a density's energy to be divided into several clusters, to account for mixed hits. The idea was that while looking for the closest higher energy density for each density (primary parent), the second closest higher energy density (secondary parent) would also be saved. Then, after clustering, all the densities whose primary and secondary parents did *not* have the same cluster ID were considered mixed densities. The energy of a mixed density was then divided to the two clusters based on the distance to its parents. See Appendix A.1.2 for the code for finding mixed hits (note that as this code was written before the end of the project, so the exact formatting is outdated).

The addition of this version of mixed hit handling did not work too well; although admittedly, my analysis was also lacking. I cannot therefore draw any conclusion on how well this exact method works, but from my analysis I did not see any significant improvement.

3.4.5 Other modifications

Float coordinates: I use float coordinates for densities, as an EcalHit's position is given in float, and this produced slightly better results than casting the position to double before performing operations.

$\boldsymbol{d_c}$: For the one-layer case, I did not use any explicit d_c. All the hits with the same xy-coordinates were placed in the same density.

3.4.6 Further improvements

As previously mentioned, the reclustering of energy comes with some caveats; it would be good to find a way to reduce the amount of merged clusters without increasing overcounting. Also, to achieve higher energy purity, it might be necessary to find a way to handle mixed hits, as right now a hit can still only belong to one cluster. This would be especially relevant when looking more at events with more than two electrons.

When I started working with the project, I had not worked a lot with C++ and pointers were still kind of a mystery to me. This led me to create a copy of the `WorkingCluster` files, which store `EcalHits` as pointers. This copy is called `WorkingEcalCluster` and stores hits as normal objects instead. The functionality is basically the same with some improvements: removing `EcalGeometry` as it is not needed, and adding a parameterless constructor. It would be great to merge these changes into the probably more efficient `WorkingCluster` and eliminate `WorkingEcalCluster`. This will probably require some tuning in `EcalClusterProducer` and `CLUE`.

3.5 CLUE3D

In the first implementation of CLUE, I merged all layers into one. However, there could be potential improvement in performing CLUE separately on the different layers, and then connecting the clusters between different layers. An algorithm for this exists, called CLUE3D, but is not as well documented as the 2D method; the only information we could find was the code for it in CMS-SW [2]. I also started with this very late in the project and did not have time to do much more than make a draft.

3.5.1 Several layers

It is possible to use CLUE on 1-17 (actual number of ECal layers) layers. If the user asks for more than one layer, the distance between the first and last hit is divided into the number of layers, to get the size of a layer. If the user chooses 17 layers (or anything below 1 or above 17), the actual ECal dimensions are used.

The hits are sorted in increasing z. The z-coordinate of the first hit is saved as the layerZ variable. The hits are then looped through. Hits are saved to the first layer, until a hit has a z-coordinate higher than layerZ plus the size of the layer. The layer is then incremented, a new layerZ saved, and the following hits will get added to this layer instead. This is continued until all hits have been placed in a layer.

There is an additional clause when incrementing the layer that keeps track of the true layers; this is to ensure that a new custom layer is not started in the middle of an actual ECal layer, but rather in the air between these. The true layers are also being kept track of so that we can later create centroids for the first layer. The first layer centroids are created when converting to WorkingEcalClusters; for each cluster, any hits that have a z-coordinate that belongs in the first (true) layer will be added to a separate cluster. These first layer centroids can then be compared to TS tracks to determine if a cluster is accurate or not.

While creating the layers, the minimum seed separation distance of each layer is set to the containment radius of that layer, and the seed threshold is set to half the energy of the highest energy density. This is based on absolutely nothing and should be revised.

When grouping the hits into densities, d_c is used to ensure a density does not just correspond to one hit. Each layer is divided into boxes with side d_c, and each box (that has at least one hit in it) becomes a density, containing the hits with xy-coordinates in that box. The xy-coordinates of the density is the middle point of the box, and the z-coordinate is the average of all the z-coordinates of the hits in the density.

3.5.2 3D clustering

While clustering is performed on a layer (if the number of layers > 1), the seeds for the clusters are saved. After the clustering is finished, the seeds' properties are overwritten with their cluster's properties (e.g. the total energy of the seed is initially the energies of the hits included in that density, but it is now overwritten with the total energy of the cluster). Another option would be to create a new density for the cluster instead.

After all layers have been clustered, the layers are looped through. For each seed (cluster) in a layer, the seeds in the previous and next layer are looked at, and similarly to the 2D case, the closest higher energy one (if any) is saved as this seed's parent.

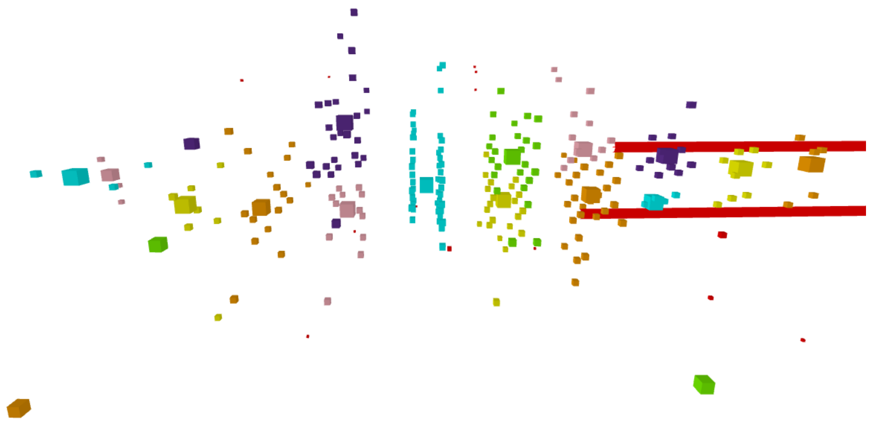
After this, the same clustering method is performed on the layer seeds as for the 2D case, to connect them into large clusters.

Rather than this rigid, layer-by-layer clustering, it would probably be better to compare the seeds' distance to each other in a theta-z-plane instead, where theta would be the angle from origin to this hit.

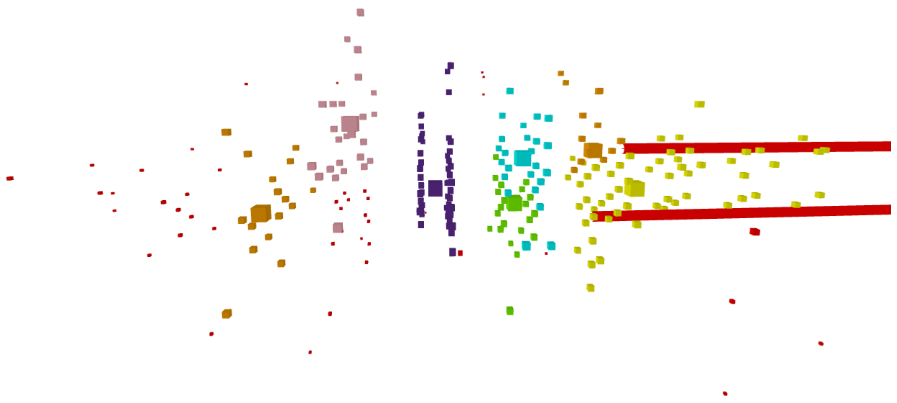
3.5.3 Results

I do not have similar analytical results for the layer and 3D clustering as for the 2D case, as I did not have time for this. I have initially analyzed the results by looking at the events in LDMX-VIS. See Figure 21a for an example of layer clustering for 17 layers for a two-electron event; here the different colors represent different clusters (most colors are reused three times), where the smaller boxes are rechits and the larger boxes the cluster centroids. The results are passable; many layers have two centroids, but some layers have only one, and especially in the further back layers, there are many clusters that are just one hit. This is because of the seed energy threshold being set to half of the highest energy hit in the layer, and in the deeper layers, hits have lower and more similar energies, so many hits are over the threshold; the hits are also far away from each other, so a density is likely to be at least δ_c away from its parent density. Potential improvements would be to increase d_c further in the ECal to make densities larger, and seeing over ρ_c and δ_c choices (i.e. parameter tuning).

Figure 21b shows the same event after the current 3D clustering method. Here, a lot of parameter tuning is needed; many clusters from the layer clustering get demoted to outliers and become unclustered hits (shown as smaller red



(a)



(b)

Figure 21: A two-electron event after a) layer clustering and b) 3D clustering.

boxes), and barely any merging happens. Here, the parameters have been set to random hardcoded values, so there is much room for improvement.

3.5.4 Potential improvements

As mentioned, I did not get very far in making the 3D clustering work well, so there are many potential improvements (most previously mentioned).

- **Parameter tuning**: The first improvement would be to find good values for all the CLUE parameters, see Table 1. For the layer clustering, the parameters need to be set separately for each layer, as hits are further away deeper in the ECal, and the energy of hits also generally decreases. For the 3D clustering, similar things should be considered.

- **Verification of the layers**: While my method of finding layers seems to work fine, I have only tried it with a few different numbers of layers; more extensive testing would be needed. It might also be easier to calculate layer dimensions once and then reuse that instead of having to recalculate layers each time CLUE is run.

- **Seed or centroid**: For the 3D clustering, I have initially used the cluster seeds to connect the 2D clusters. It might be better to use the cluster centroids instead.

- **Increasing d_c**: d_c is the same for all layers, but hits are further apart further in the ECal. It might make sense to increase d_c further in so that densities are not just one hit.

4 Summary

In this project, I have worked with clustering of reconstructed hits in the electronic calorimeter, with the main goal of achieving electron counting. Initially, I set up a visualisation tool in Phoenix and an analyzer for transforming ECal event data into JSON files to use in said visualiser. After this, I tried out several different algorithms for electron clustering, working mainly with the initial (already existing) algorithm, and CLUE.

While the initial algorithm and CLUE produced similar results when it came to the number of electrons (around 1.8 for two electrons) and energy purity (around 90% for two electrons), CLUE outperformed the initial algorithm by far when it came to the percentage of clusterless hits: for two electrons, CLUE had around 7%, while the initial algorithm had around 46%. In light of this, CLUE's performance is better, as it is "easier" for clusters to be energetically pure if they contain much fewer hits.

I also found CLUE much easier to work with, as it is modular. The clustering process involves several steps and thus has more potential for modifications, in comparison to the initial algorithm, where the only modification opportunity is in the weight. With future improvements such as reclustering, mixed hit handling and CLUE3D, clustering with CLUE has much potential to improve even further.

References

[1] ATLAS Collaboration. "Topological cell clustering in the ATLAS calorimeters and its performance in LHC Run 1". In: *The European Physical Journal C* 77.490 (2017). DOI: `https://doi.org/10.1140/epjc/s10052-017-5004-5`.

[2] CMS-SW. *Pattern Recognition by CLUE3D.* URL: `https://github.com/cms-sw/cmssw/blob/master/RecoHGCal/TICL/plugins/PatternRecognitionbyCLUE3D.cc#L680` (visited on 08/14/2024).

[3] pbutti. *LDMX Event Display.* URL: `https://github.com/pbutti/ldmx-event-display` (visited on 08/19/2024).

[4] Phoenix. *Convert GDML Geometry to ROOT and GLTF.* URL: `https://github.com/HSF/phoenix/blob/main/guides/developers/convert-gdml-to-gltf.md` (visited on 08/14/2024).

[5] Phoenix. *Event Data Format.* URL: `https://github.com/HSF/phoenix/blob/main/guides/developers/event_data_format.md` (visited on 08/16/2024).

[6] Phoenix. *Set up Phoenix for an experiment.* URL: `https://github.com/HSF/phoenix/blob/main/guides/developers/set-up-phoenix.md` (visited on 08/14/2024).

[7] M. Rovere, Z. Chen, A. Di Pilato, F. Pantaleo, and C. Seez. "CLUE: a fast parallel clustering algorithm for high granularity calorimeters in high-energy physics". In: *Frontiers in big Data* 3 (2020). DOI: `https://doi.org/10.48550/arXiv.2001.09761`.

A Code

A.1 CLUE

A.1.1 Reclustering based on energy

In CLUE::clustering:

```
do { ...
    if (energyOverload) {
        deltacMod = deltacMod/1.1;
        if (debug_) std::cout << "Energy overload, new deltacmod: " << deltacMod
        ↪   << std::endl;
        energyOverload = false;
    }
    clusteringLoops_++;
    ...
    for (int j = 0; j < densities.size(); j++) {
        int i = densities[j]->index;
        if (deltacMod != deltac_ && mergedDensities[densities[i]->clusterId]
            && floatDist(densities[i]->x, densities[i]->y,
            ↪   eventCentroid_.centroid().Px(), eventCentroid_.centroid().Py()) <
            ↪   centroidRadius) {
          isSeed = densities[i]->totalEnergy > rhoc_ && densities[i]->fDelta >
          ↪   deltacMod;
        } else isSeed = densities[i]->totalEnergy > rhoc_ && densities[i]->fDelta
        ↪   > deltac_;
        ...
        densities[i]->clusterId = -1;
        ...
    }
    while (clusterStack.size() > 0) {
        auto& d = densities[clusterStack.top()];
        clusterStack.pop();
        auto& cid = d->clusterId;
        for (const auto& j : followers[d->index]) { // for indices of followers
        ↪   of d
          auto& f = densities[j];
          ...
          clusterEnergies[cid] += f->totalEnergy;
          if (reclustering_ && clusterEnergies[cid] > maxEnergy && deltacMod >
          ↪   0.5 && clusteringLoops_ < 100) {
            mergedDensities[cid] = true;
            if (!energyOverload && clusteringLoops_ == 99) std::cout << "Merged
            ↪   clusters, max cluster loops hit" << std::endl;
            energyOverload = true;
            if (clusteringLoops_ != 1) goto endwhile; // don't break on first
            ↪   loop to save initial cluster number
          }
          ...
        }
    }
    endwhile:;
} while (energyOverload);
```

A.1.2 Mixed hits

Presented is the method in `CLUE.h` that found mixed hits after each density has received both a primary and secondary parent, and clustering has been performed. Note that the mixed hit functionality, including this method, has since been removed, but it is presented here in case someone wants to attempt something similar.

```
void findMixedHits(std::vector<std::shared_ptr<Density>>& densities,
↪  std::vector<std::vector<std::shared_ptr<Density>>>& densityClusters) {
    if (debug_) std::cout << "Looking for mixed hits" << std::endl;
    for (int i = 0; i < densities.size(); i++) {
      int& parentIndex = densities[i]->followerOf;
      int& secondParentIndex = densities[i]->secondFollowerOf;
      if (densities[i]->clusterId != -1 && secondParentIndex != -1 &&
      ↪  densities[secondParentIndex]->clusterId != -1 &&
      ↪  densities[i]->clusterId != densities[secondParentIndex]->clusterId) {
        densities[i]->secondClusterId = densities[secondParentIndex]->clusterId;
        if (debug_) std::cout << "  Mixed density with index " <<
        ↪  densities[i]->index << "; c1: " << densities[i]->clusterId << "; c2:
        ↪  " << densities[i]->secondClusterId << std::endl;
        // Calculate dist to parent vs second parent
        float dc1 = floatDist(densities[parentIndex]->fx,
        ↪  densities[parentIndex]->fy, densities[i]->fx, densities[i]->fy);
        float dc2 = floatDist(densities[secondParentIndex]->fx,
        ↪  densities[secondParentIndex]->fy, densities[i]->fx,
        ↪  densities[i]->fy);
        // Alt calculate dist to first cluster seed and second cluster seed --
        ↪  gives slightly worse results
        // float dc1 = floatDist(densityClusters[densities[i]->clusterId][0]->fx,
        ↪  densityClusters[densities[i]->clusterId][0]->fy, densities[i]->x,
        ↪  densities[i]->y);
        // float dc2 =
        ↪  floatDist(densityClusters[densities[i]->secondClusterId][0]->fx,
        ↪  densityClusters[densities[i]->secondClusterId][0]->fy,
        ↪  densities[i]->x, densities[i]->y);
        if (debug_) std::cout << "    Distance to c1: " << dc1 << "; c2: " << dc2
        ↪  << std::endl;
        if (dc1 + dc2 > 0)  {
```

```
        densities[i]->c1p = 1-dc1/(dc1+dc2);
        densities[i]->c2p = 1-dc2/(dc1+dc2);
      }
    }
  }
}
```

A.2 Topoclustering

TemplatedEcalClusterFinder

This code was the implementation of ATLAS topoclustering; it has since been removed, but is presented here for completeness.

```
/*
   TemplatedEcalClusterFinder
   */

#ifndef ECAL_TEMPLATEDECALCLUSTERFINDER_H_
#define ECAL_TEMPLATEDECALCLUSTERFINDER_H_

#include <math.h>
#include <algorithm>
#include <unordered_map>
#include <iostream>

#include "Ecal/WorkingEcalCluster.h"

#include <iostream>

namespace ecal {

template <class WeightClass>

class TemplatedEcalClusterFinder {
 public:

  void createCentroid(const ldmx::EcalHit& eh) {
    centroids_.insert({eh.getID(), WorkingEcalCluster(eh)});
  }

  static bool compHits(const ldmx::EcalHit& a, const ldmx::EcalHit& b) {
    return a.getEnergy() > b.getEnergy();
  }

  void recursive_clustering(const ldmx::EcalHit& current_hit, WorkingEcalCluster&
  ↪  centroid, std::vector<int>& path) {
```

```
  depth_++;
  for (auto& hit : hits_) {
    if (hit.getID() == current_hit.getID()) continue;
    if (std::find(path.begin(), path.end(), hit.getID()) != path.end())
    ↪  continue;
    double wgt = wgt_(current_hit, hit);
    if (wgt < neighbourWeight_) {
      if (hit.getEnergy() >= cellFilter_) {
        path.push_back(hit.getID());
        transitionWeights_.push_back({nClusters_, wgt});
      }
      if (hit.getEnergy() >= seedThreshold_) {
        // hit is seed, merge clusters
        // find centroid
        auto it = centroids_.find(hit.getID());
        if (it != centroids_.end()) {
          auto& mergeCentroid = it->second; // WorkingEcalCluster
          int size = mergeCentroid.getHits().size();
          for (const auto& mhit : mergeCentroid.getHits()) {
            path.push_back(mhit.getID());
          }
          centroid.add(mergeCentroid);
          mergeCentroid.clear();
          nClusters_--;
          // if centroid only has seed in it, explore neighbours
          // otherwise seed has already been looped through
          if (size == 1) recursive_clustering(hit, centroid, path);
        }
      } else if (hit.getEnergy() >= growthThreshold_) {
        centroid.add(hit);
        recursive_clustering(hit, centroid, path);
      } else if (hit.getEnergy() >= cellFilter_) {
        centroid.add(hit);
      }
    }
  }
  loops_++;
}

void cluster(std::vector<ldmx::EcalHit>& ecalHits,
                        double seed, double growth, double filter, double
                        ↪  neighbour_wgt) {
  hits_ = ecalHits;
  seedThreshold_ = seed;
  growthThreshold_ = growth;
  cellFilter_ = filter;
  neighbourWeight_ = neighbour_wgt;
  // Sort after highest energy
```

```cpp
    std::sort(hits_.begin(), hits_.end(), compHits);
    // Find seeds
    for (const auto& hit : hits_) {
      if (hit.getEnergy() >= seedThreshold_) {
        createCentroid(hit);
      } else break;
    }
    nClusters_ = centroids_.size();
    loops_ = 0;
    for (auto& [id, centroid]: centroids_) {
      if (centroid.empty()) continue;
      std::vector<int> path;
      path.push_back(centroid.getHits()[0].getID());
      recursive_clustering(centroid.getHits()[0], centroid, path);
    }

    for (auto& [id, centroid]: centroids_) {
      if (!centroid.empty()) {
        finalClusters_.push_back(centroid);
      }
    }
  }

  int getNSeeds() const { return nseeds_; }

  int getNLoops() const { return loops_; }

  std::vector<std::pair<int, double>> getWeights() const { return
  ↪  transitionWeights_; }

  std::vector<WorkingEcalCluster> getClusters() const { return finalClusters_; }

 private:
  WeightClass wgt_;
  double finalwgt_;
  int nseeds_;
  int loops_;
  int depth_;
  int nClusters_;

  double neighbourWeight_;
  double seedThreshold_;
  double growthThreshold_;
  double cellFilter_;

  std::vector<std::pair<int, double>> transitionWeights_;
  std::vector<ldmx::EcalHit> hits_;
  std::unordered_map<int, WorkingEcalCluster> centroids_;
  std::vector<WorkingEcalCluster> finalClusters_;
};
}  // namespace ecal

#endif
```