CLEO III Cathode Hit Calibration

Dawn K. Isabel

Department of Electrical and Computer Engineering,
Wayne State University, Detroit, MI, 48202

Abstract

The drift chamber cathodes for CLEO III are located at the outer cylindrical surface of the drift chamber, and are used to determine the position of a track along the beam axis. We describe the software that has been created to convert the raw output from the detector to a form that can be used by track reconstruction algorithms. This calibration software is written for use in Suez, the C++ job control and user interface for CLEO III. We implement the Strategy design pattern to allow flexibility in choosing calibration algorithms and facilitate future additions to the code.

Introduction

The drift chamber (DR) cathodes [1] for CLEO III are located at the outer cylindrical surface of the drift chamber, and are segmented into octants in ϕ (Figure 1). Each octant is further divided into 192 cathode pads along the beam axis. When a charged particle passes through the last layer of the DR, the positive charge liberated by the ionization of the drift chamber gas is collected by the cathodes, and can be spread over one or more pads in an octant. Each pad is read out separately by the data acquisition system, and those with nonzero ADC values are considered to be "raw hits". A clustering algorithm is applied to find the raw hits associated with a single

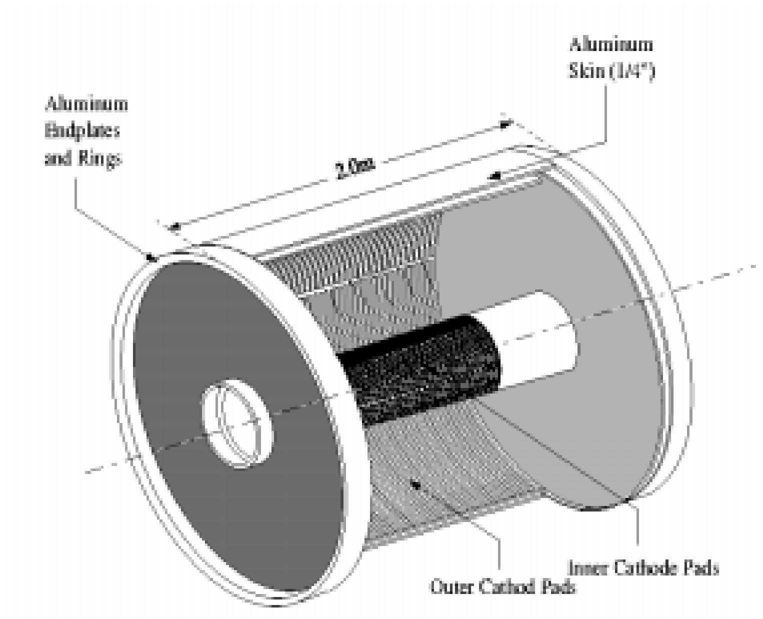

FIGURE 1. CLEO II DR cathodes. There is only an outer cathode layer in CLEO III.

track, and this group of raw hits is called a "calibrated hit" (Figure 2). Once the raw hits that compose a calibrated hit are identified, the centroid, or Z position, of this cluster is calculated according to various algorithms.

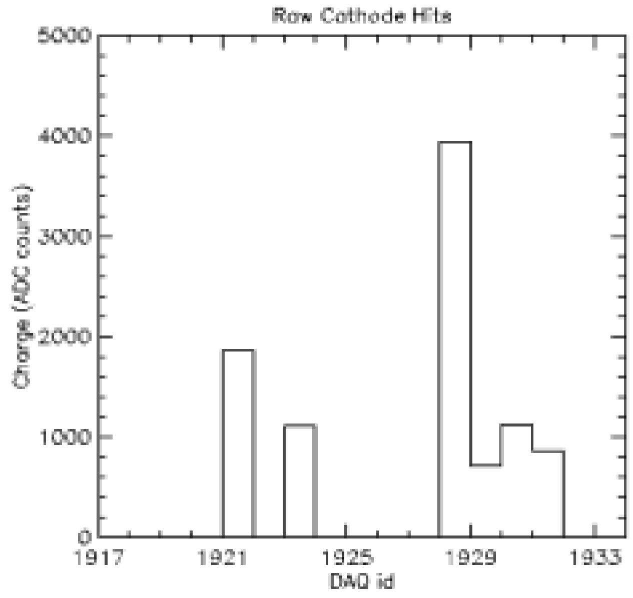

FIGURE 2. Raw cathode hits. A clustering algorithm will be applied to identify calibrated hits.

For a group of raw hits to be considered a cluster, the following conditions must be met:

- All charges must be greater than a given cluster threshold.
- The peak charge in the cluster must be greater than a given peak threshold.
- If there are charges on two or more pads, the pads must be adjacent.

These conditions are the same as the existing algorithm for CLEO II. It has not yet been determined how to handle dead channels adjacent to found clusters.

When a cluster is found, the following parameters can be used to describe it and to construct a calibrated cathode hit:

- Number of pads in cluster
- Peak charge
- Total charge
- Z position[1]
- Width[2]

[1] The z position (or centroid) of a cluster is determined for our purposes in one of two ways. Suppose we have a cluster of N pads, each pad has charge Q_i ($i = 1,N$), the width of a pad is Δz, and the z coordinate of the center of the *i*th pad is $z_i = (i-1/2)\Delta z$. A center of gravity centroid—a weighted average using the charges on the pads as weights—can be written as

$$\bar{z} = \frac{\Delta z}{Q_{tot}} \sum_{i=1}^{N} Q_i (i - \frac{1}{2})$$

where the origin is the edge of the first strip. A second algorithm where all weights are equal replaces Q_i/Q_{tot} with 1/N.

The Conceptual Model and Suez

Suez [2] is the CLEO III job control and user interface for data analysis in C++. It provides a Tcl command-line interface to interact with processors and producers written by the user. These processors and producers are given access to data via the "Frame", which is a representation of all the information about CLEO at any given time. A Frame can be constructed by collecting together the data that is current at a particular time. The data is grouped by "lifetime"—groups sharing a lifetime are called "Records". A time-ordered set of Records that describe the same part of the state, but at different times, is called a "Stream".

Usually the user is interested only in new Records in certain Streams. For example, if a user wants to stop at each event, the events are "Stops", while the event stream is the "Active" Stream since it provides the Stops.

Job Control: Processors, Producers and Proxies

Actions to be taken by the user (like executing some code) are bound to a Stream. If the Stream is active, the Action will be executed. Using this model, a processor is the container of these Actions. A processor is triggered by a Stop to perform some algorithm, and will then insert the results into the Frame. This means that processors stop on every event in the stream. Producers, on the other hand, only execute when asked, and put algorithms into the Frame. This is accomplished by registering proxies [3] with the Frame. A proxy is a placeholder for objects that haven't yet been created. The algorithms contained in the proxies will run or return results only when asked, which saves time.

Calibrated Cathode Hit Software

To create calibrated cathode hits from the clustering of the corresponding raw hits, software needed to be written in C++ that could be used in Suez. This required several different things: first, a class that defines what a calibrated cathode hit is; a processor to aid in the debugging process; a producer and proxy to call the proper algorithms for calculations; and finally two families of classes, Centroid and Cluster, that contain the algorithms.

CalibratedCathodeHit

The `CalibratedCathodeHit` class (Figure 3) contains all the information vital to later use of the data for track reconstruction. Each data member has a corresponding access function with the same name so that private data members can be viewed, and there are additional functions for changing the value of the corresponding data member.

---

[2] The r.m.s. width of a cluster describes the resolution of the centroid, and is used in track reconstruction. Width calculations are dependent on the z position of a cluster (given by one of the previous expressions):

$$Width = \sqrt{\frac{1}{3Q_{tot}\Delta z}\sum_{i=1}^{N} Q_i[(i\Delta z - \bar{z})^3 - ((i-1)\Delta z - \bar{z})^3]}$$

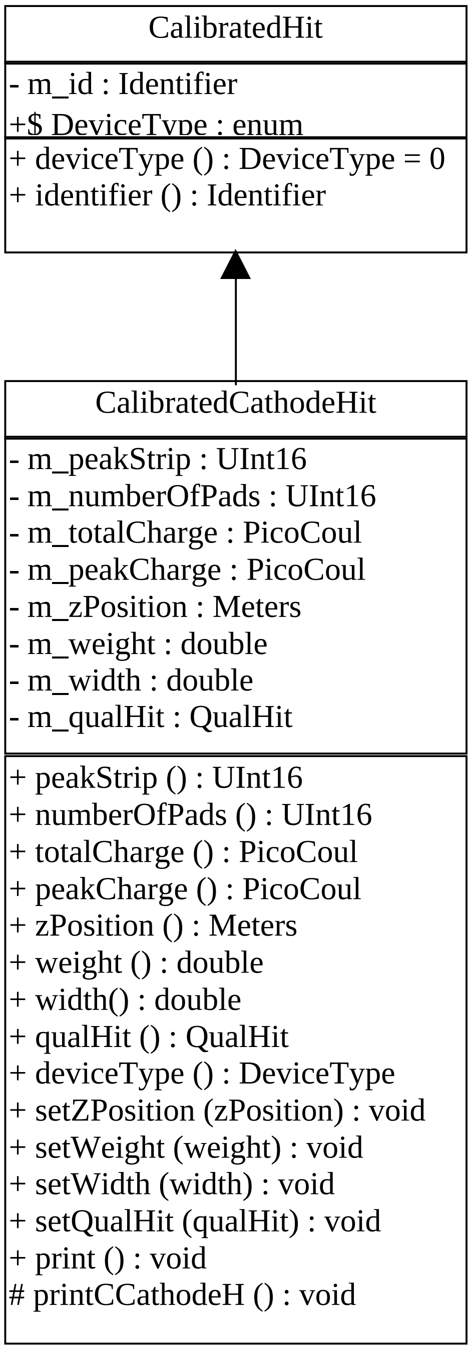
CalibratedHit
- m_id : Identifier
+$ DeviceType : enum
+ deviceType () : DeviceType = 0
+ identifier () : Identifier
CalibratedCathodeHit
- m_peakStrip : UInt16
- m_numberOfPads : UInt16
- m_totalCharge : PicoCoul
- m_peakCharge : PicoCoul
- m_zPosition : Meters
- m_weight : double
- m_width : double
- m_qualHit : QualHit
+ peakStrip () : UInt16
+ numberOfPads () : UInt16
+ totalCharge () : PicoCoul
+ peakCharge () : PicoCoul
+ zPosition () : Meters
+ weight () : double
+ width() : double
+ qualHit () : QualHit
+ deviceType () : DeviceType
+ setZPosition (zPosition) : void
+ setWeight (weight) : void
+ setWidth (width) : void
+ setQualHit (qualHit) : void
+ print () : void
# printCCathodeH () : void

FIGURE 3. `CalibratedCathodeHit` class. Note the inheritance from `CalibratedHit`.

In addition, the `CalibratedCathodeHit` class inherits from the abstract base class `CalibratedHit` (as do all calibrated hit classes), which provides the additional data member `m_id` and corresponding `identifier()` access function, as well as the `deviceType()` function, which returns the name of the device being used (in this case, the device name is "`Cathode`").

CalibratedCathodeHitProcessor

In the `CalibratedCathodeHit` class, the processor performs several important tasks:

- Creates histograms and Ntuples
- Prints significant raw data to the screen
- Prints `CalibratedCathodeHit` partnered with raw data for that hit

The processor is primarily used to detect and pinpoint errors during debugging, since all the relevant data is displayed before and after calibration.

CalibratedCathodeHitProducer/CalibratedCathodeHitProxy

Before the processor can manipulate the calibrated cathode hits, they must be generated from the existing raw cathode data. This is accomplished using the `CalibratedCathodeHitProducer`. When the processor asks for `CalibratedCathodeHits`, the producer invokes the `CalibratedCathodeHitProxy`, which houses the algorithm that creates the `CalibratedCathodeHits`. Once the producer creates the newly calibrated cathode hits, it inserts them into the Frame. The producer also creates link objects that connect each `CalibratedCathodeHit` to the associated raw hits. The user can then retrieve the associated raw hits with the use of a navigator object [4].

The Strategy Pattern

Several methods exist for clustering raw hits and calculating the centroid. To make the DR cathode software as flexible as possible, the Strategy design pattern [3] is used. To encapsulate each algorithm in a family makes them interchangeable, so that they are independent of the clients who use them. This gives the user the ability to mix and match algorithms in any combination at run-time.

The Strategy pattern relies on inheritance for its flexibility. A private data member of the base class type is assigned to an instance of a subclass at run-time, and a common interface allows objects to be interchangeable. This facilitates the addition of new algorithms: all that is needed is a new subclass. The comparison of algorithms is made easier, because the choice of algorithm is decided at run-time.

Implementing the Strategy Pattern

The Centroid and Cluster families (Figure 4) are implementations of the Strategy design pattern. The Centroid and Cluster classes are both abstract base classes containing one pure virtual function. This means that an instance of either class cannot be instantiated. Instead, derived classes are created that inherit the public data and member functions from their base class, and override the pure virtual functions with their own unique algorithms.

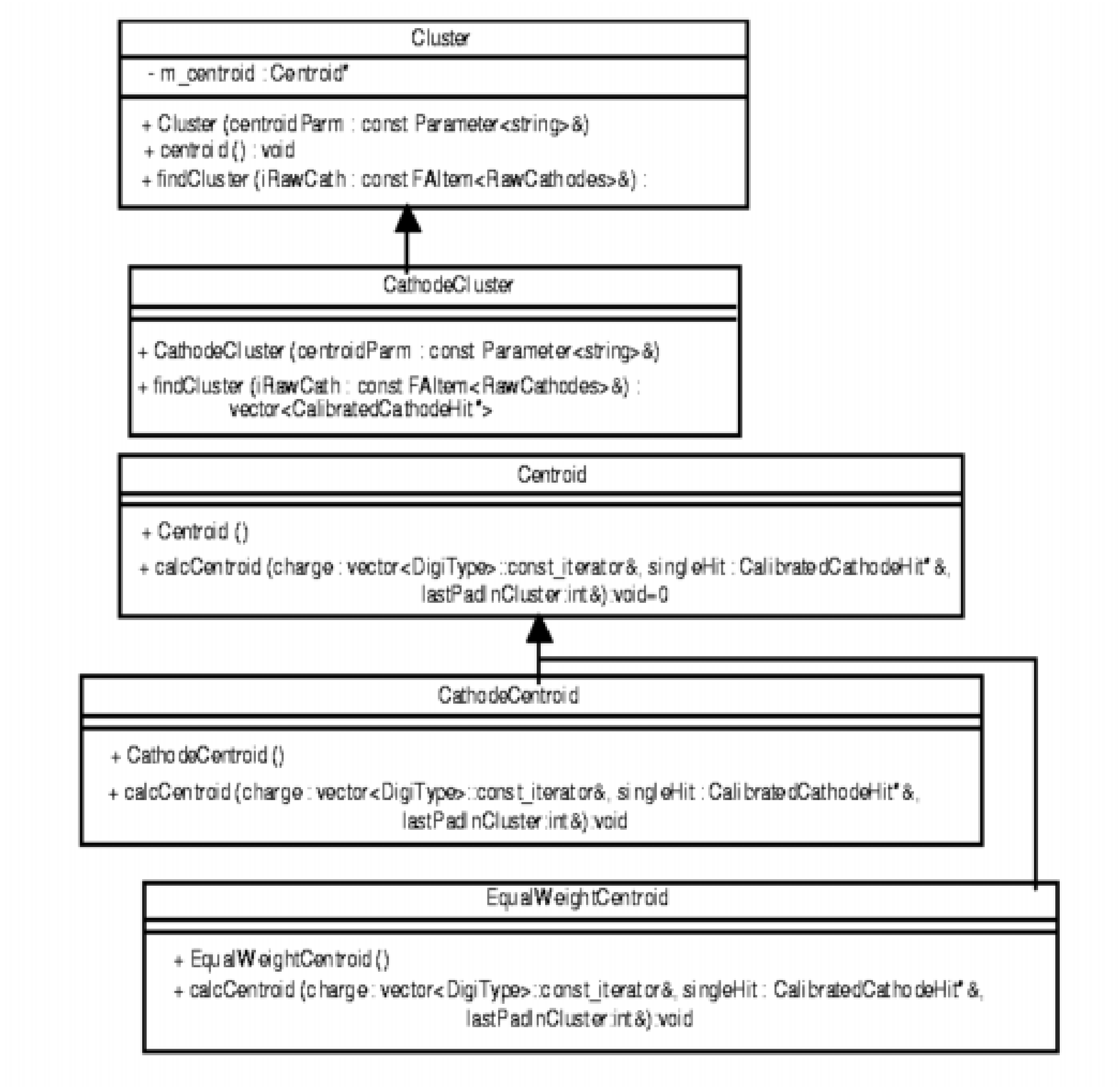
Cluster
- m_centroid : Centroid*
+ Cluster (centroidParm : const Parameter<string>&)
+ centroid () : void
+ findCluster (iRawCath : const FAItem<RawCathodes>&) :
CathodeCluster
+ CathodeCluster (centroidParm : const Parameter<string>&)
+ findCluster (iRawCath : const FAItem<RawCathodes>&) : vector<CalibratedCathodeHit*>
Centroid
+ Centroid ()
+ calcCentroid (charge : vector<DigiType>::const_iterator&, singleHit : CalibratedCathodeHit*&, lastPadInCluster:int&):void=0
CathodeCentroid
+ CathodeCentroid ()
+ calcCentroid (charge : vector<DigiType>::const_iterator&, singleHit : CalibratedCathodeHit*&, lastPadInCluster:int&):void
EqualWeightCentroid
+ EqualWeightCentroid ()
+ calcCentroid (charge : vector<DigiType>::const_iterator&, singleHit : CalibratedCathodeHit*&, lastPadInCluster:int&):void

FIGURE 4. Cluster and Centroid class diagrams. Derived classes point to their base class with arrows.

To implement the Strategy Pattern in Suez, we make use of run-time parameters to alter the behavior of processors or producers. The user gives the parameters to the producer through the Suez command-line interface. The parameters are bound to the producer, which uses them to select subclasses of Cluster and Centroid to instantiate. The parameters can be changed, however, at any point in the session. To assist users, help menus were constructed for each parameter, and can be accessed in Suez at run-time.

Results and Conclusions

Figure 5 shows the result of using the new cathode calibration software in conjunction with existing RawData software in Suez. The histogram was produced from an Ntuple made by the `CalibratedCathodeHitProcessor`, and depicts the raw cathode hits and their charge. The overlaid curves show Gaussians with means and widths equal to those of the corresponding cluster. The plot distinctly shows the difference in results from the two centroid algorithms. Having easy access to a number of algorithms gives the software a great deal of added flexibility.

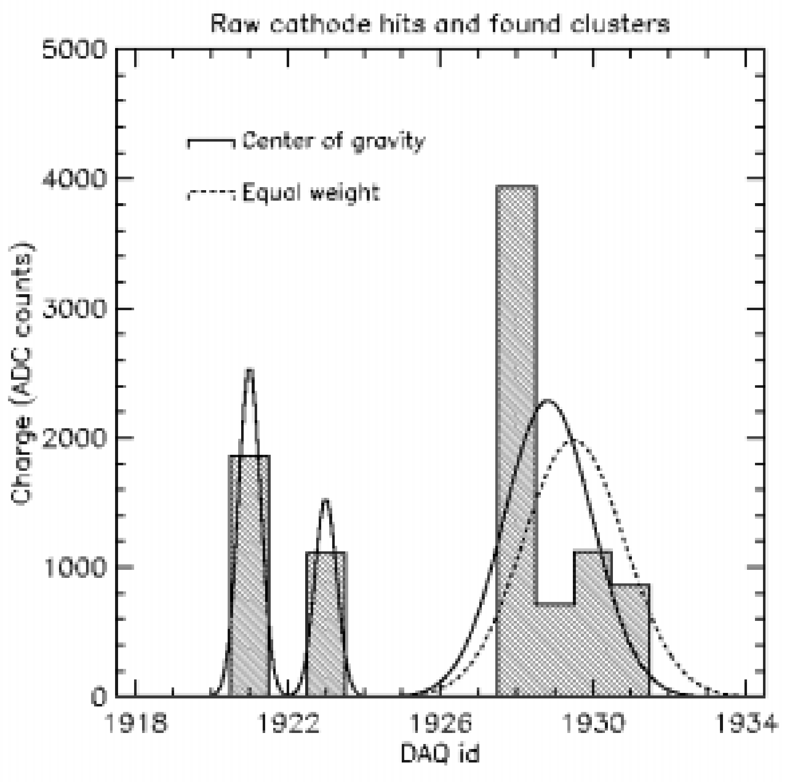

FIGURE 5. Raw cathode hits and found clusters.

This new software offers several important benefits. The `CalibratedCathodeHitProcessor` gives the user the ability to view the `RawCathodes` and `CalibratedCathodeHits` as the conversion is performed, and simultaneously creates a historgram and Ntuple of the `RawCathode` data. The parameters used in the

`CalibratedCathodeHitProducer` allow the user to defer selection of the particular algorithms until run-time. This flexibility is ideal for testing new algorithms and comparing results between existing ones. The use of Tcl parameters also allows the user to mix and match combinations of cluster and centroid algorithms.

In addition to refining both the clustering and centroid calculation algorithms, future directions include extending the applicability of the Centroid and Cluster families. The abstract base classes can be modified to be independent of the cathode software, so that other devices that need clustering algorithms, like the silicon vertex detector, could also make use of the same abstract classes. It is possible that this may be accomplished using templated classes.

Acknowledgments

I would like to gratefully acknowledge Prof. Alan Weinstein and Mr. Werner Sun of the California Institute of Technology, who developed and supervised this REU project and gave their time and resources to educate me in the physics of cathodes and the applications of C++ and Suez. I would also like to thank Dr. Adam Lyon of the University of Rochester for his time and endless patience with my class diagrams, and Profs. Giovanni Bonvicini, David Cinabro, and Rodney Greene of Wayne State University for their help in preparing for this program. This work was supported by the California Institute of Technology, Division of Physics, Mathematics and Astronomy, the National Science Foundation grant PHY-9310764 and NSF REU grant PHY-9731882.

References

1. Y. Kubota *et al.* (CLEO Collaboration), Nucl. Instr. Methods Phys. Res. A**320**, 66 (1992).
2. http://www.lns.cornell.edu/restricted/CLEO/CLEO3/soft/Suez/Doc/suez/index.html
3. Gamma, Erich *et al*., Design Patterns. Addison-Wesley. (1995)
4. http://www.lns.cornell.edu/restricted/CLEO/CLEO3/soft/Tracking/TrkInfrastruct/trkinf2.doc