

Calorimeter Review Summary

Jim Kowalkowski, Marc Paterno

1 Introduction

This is a brief note meant to describe the changes to the calorimetry code that we recommend. Accompanying this note is a class diagram indicating how we think the system should fit together. This diagram is by no means complete. It is designed to serve as a higher-level view of the system components.

2 Overview

2.1 Names

The diagram uses new class names. Many of these classes have a direct relationship to the classes in the existing code; sometimes the equivalence is nearly exact, and sometime the equivalence is very approximate.

PhysicsCalorData	TOWER_COLLECTION
CalorData	TOWE_BANK_DRIVER
CalorTower	TOWE_BASE
PhysicsTower,Clusterable	CALTOWER
Locations	EVENT_ANNULUS_LOCATIONS
TowerThetaContainer	Detector_Boundaries
CalorTowerKey	TOWE_Key
CellKey	TOWE_Key
ElectronicsKey	Bank_Cal_Key
CalorGeometryNode	Cal

2.2 Subsystems

The diagram shows four basic subsystems:

1. Creation of the CalorData objects. This is the D-Banks converted to Energy for each of the towers in the calorimeter. CalMod carries out this process. CalorData is meant to be stored in the event, it is mostly just a fancy container that allows direct access to a tower given (ieta, iphi). The persistent form of CalorData could be (but need not be) TOWE. Creating CalorData from the D-Banks is fairly straightforward – make an instance of a TowerTypeX, fill it, and insert it into the CalorData container. Restoring the CalorData from disk using the TOWE format will probably require a “factory” similar to the way it is currently done in order to automate the creation of the TowerTypeX objects. The diagram shows two forms of iteration that should be made available:

standard iteration through all the CalorTowers, and special iteration through CalorTowers from a specific detector such as “Central” (see DetIterator). In the new code, CalorData should allow one to get a CalorTower, not manipulate it directly as the current TOWE_BANK_DRIVER does.

2. The reconstructed calorimeter or PhysicsCalorData. There is one PhysicsCalorData instance created for each interesting combination of primary vertex, input source, and “calculator” (to be defined later). The PhysicsCalorData instances can be stored in the event (EventRecord). The persistent form of the PhysicsCalorData could be just the parameters used to create it (AbsParms, the identifier of the CalorData object, the z position of the vertex, etc.), and not the actual PhysicsTowers. This should be done if rebuilding it is as computationally cheap as indicating during the review. We think this may not actually be true if the input source is HEPG. The concept of a view has been introduced here to manage activities such as seed lists and clustering. It is unclear at the present moment whether or not the views need to be storable in the event. For efficiency purposes, the individual PhysicsTowers could maintain pointers to a list of CalorTowers used to generate it.
3. The creation of the PhysicsCalorData object. In the current code, there are really two types of calculators, or “summers”. The current code abstracts out a summer, used in determining the centroid of a cluster. This code is not present in the diagram and should be moved to the jet finding package. PhysicsCalorMaker controls the creation of the PhysicsCalorData. A high-level module such as JetMod should use the AC++ AbsParms or menus to determine the input source, calculator, and perhaps the vertex. The module can then create the correct instance of InputSource (HEPG or CalorData based). The module uses the input source and vertex to call “create” in the PhysicsCalorMaker, which makes a PhysicsCalorData object. The input source appears like a stream, where each call to “next” produces enough information for the PhysicsCalorMaker to produce a PhysicsTower. Each specific type of input source has a set of calculator that can be used for summing energies. Note that the process of taking Monte Carlo generated “particles” from HEPG, and creating a PhysicsCalorData object is a detector simulation process. In time, many different PhysicsCalorMakers could be provided, each of which contains a different level of detail in simulation, different tunable parameters, etc.
4. Geometry related classes. All the geometry related classes should be moved from the Calor package to a CalorGeometry package. The current Cal class needs to be changed to conform to the new geometry model. Two “keys” should be present in the new system, one for accessing towers, and one for accessing individual cells within a tower. The keys are smart enough to indicate their neighbors. A set of helper functions should be written to aid in the creation or filling of collections of neighboring keys. The diagram also shows an iterator that allows one to go through all the tower keys in the detector, or keys that pass conditions coded into a predicate.

3 Descriptions

Following is a brief description of the purpose of each class shown in the class diagram.

- *CalorData*: A container class that makes the calorimeter look like a big grid of towers, even though the actual towers in the grid are different types. Provides all interesting types of access to the towers, such as direct access by (ieta, iphi) and efficient iterating over distinct sets of towers.
- *CalorTower*: An abstract base class used to present all calorimeter towers as the same thing. This consists of methods to access data common to all towers, and virtual methods required to access tower specific data (though polymorphism).
- *CalMod*: An AC++ module that runs over the D-Banks in the calorimeter, create instances of specific CalorTowers, and inserts them into the CalorData.
- *StorableObject*: The EDM class that allows objects to be stored in the event, converted to persistent form and reconstituted from a persistent form.

- *DelIterator*: An intelligent iterator that holds a predicate. It allows the one to go through CalorTowers held within an instance of CalorData. The predicate will allow iteration only over all CalorTowers in a particular detector.
- *TowerTypeX*: There is one concrete class for each of the various tower types ($X=[0,9]$). This is very similar to the current system's TOWER_TYPE[0-9] classes. Basically it is a C++ view of the information inside TOWE.
- *PhysicsCalorData*: Another container class. This class retains all the information that was used to generate it. This information includes the source (example is CalorData) identifier, the z location of the vertex used, the input source name (algorithm name), and the input source calculator name (sub-algorithm name). The class also caches an instance of an "EventAnnulusLocations" classes based on the z location of the vertex. The Locations instance is a transient object, not meant to be stored. This container class must provide fast direct access to a particular PhysicsTower. The user must get a PhysicsTower out of the container and use the PhysicsTower interface to manipulate it. The class provides an iterator to go through all the PhysicsTowers in the PhysicsCalorData object. This is the class most consumers of calorimeter information will use to perform physics tasks, so it should provide all the necessary access mechanisms to make those tasks easy.
- *PhysicsTower*: This is a reconstructed calorimeter tower. It is very similar to the current CALTOWER class.
- *Clusterable*: The physics tower is broken up into two parts. This class represents low-level, simple quantities that can be used in clustering. We envision that, in the future, things other than PhysicsTower will be considered clusterables, such as a cluster of PhysicsTowers. Tools that manipulate clusterables can then be produced that do not need to know if the items these work with are actually a PhysicsTower or something else. This class should be useful in implementing flexible successive re-combination ("KT") jet algorithms.
- *PCView*: (PhysicsCalorView) This is a container of pointers to the PhysicsTowers. The user can sort on of these, or iterate though the elements. Using the utility function to generate it, it can actually be a standard STL list or vector of PhysicsTower pointers.
- *Locations*: This is essentially the same thing as the EventAnnulusLocations, except the user cannot interact with the elements inside at the container level. This is strictly a container of ThetaCalcs, which are calculated with a given z -vertex.
- *ThetaCalcs*: This class provides all the important trigonometric calculations of the azimuthal location of one tower (η), as measured from a given z location on the beam axis.
- *PhysicsCalorMaker*: This is an algorithm object that has the responsibility of creating a PhysicsCalorData instance given an InputSource and the z location of a primary vertex. It calls "next" on the input source, to get the input information, and creates a PhysicsTower, which it adds to the PhysicsCalorData object it is producing.
- *EnergyData*: This is the calculated energy deposited in a specific tower, denoted by (η , ϕ), used in the creation of a PhysicsTower.
- *InputSource*: This is a generic source of tower energy information. Derived classes implement the reading of the actual tower source information.
- *SourceHEPG*: This is a HEPG input source. The job here is to accumulate the energy from particles into a tower grid. The EnergyData is produce at each call of "next" for the next tower in the grid. Different subclasses of *SourceHEPG* could provide different summing methods, smearing, more or less detailed simulation, etc.
- *SourceCalorData*: This is the EnergyData information calculated from CalorData (the calorimeter).
- *HepgCalculator*: This is an abstract calculator class used to calculate the values in EnergyData for a given grid cell in the SourceHEPG grid.
- *Clump*: A particular way of calculating the EnergyData at a specific grid position.

- *Normal*: Another particular way of calculating the EnergyData at a specific grid position.
- *CalorCalculator*: This is an abstract calculator used to calculate the values in EnergyData for a given CalorTower.
- *FourVec*: This is a particular way of doing the conversion from CalorTower to EnergyData.
- *StdTower*: This is a particular way of doing the conversion from CalorTower to EnergyData.
- *MinStdTower*: This is a particular way of doing the conversion from CalorTower to EnergyData.
- *TowerThetaContainer*: This is essentially Detector_Boundaries in the current system, except that is available everywhere as a singleton and that it appears as a container. The user must ask the container for a TowerTheta object at a given ieta.
- *TowerTheta*: This is the trigonometric values associated with a particular ieta, as measured from $z = 0$.
- *CalorGeometryNode*: This is the main geometry system node that replaces the Cal class. This class directly supports looking up a tower or cell given a point in space.
- *CalorTowerKey*: This class knows about a tower position and it's relationship to other towers
- *CalorKeyIterator*: A smart iterator that allows one to iterator over all the tower keys in the calorimeter, or a specific set of keys given a predicate.
- *CellKey*: This class allows access to a particular cell in the calorimeter. It also allow location of the center of the cell.
- *Index*: This is just an (ieta,iphi) pair.
- *ElectronicsKey*: This is the geometry replacement for Bank_Cal_Key.

4 EDM Considerations

Since the CDF EDM is not expected to be complete until after the recoding recommended here, several classes and functions should be created to simulate the new EDM classes. The temporary classes include StorableObject, StorablePointer<T>, ObjectHandle<T>. A set of functions should be written to invoke the various methods of StorableObject and create StorableObjects from the event instead of just searching the event for the desired object. An example of a creation function could be “retrieveCalorData(key, event)” which returns an ObjectHandle<CalorData> instance. In the new EDM, the job of retrieveCalorData() would be to search the event for the CalorData object return it. If the object does not exist, it would be created, stored in the event, and returned. In the current system, the job will be to look for the banks required to make the CalorData instance and then generate it.

We think it would be useful to consult with Rob Kennedy on features of the Event Model, in order to minimize the amount of re-design and re-coding necessary to adapt to the new Event Model.

5 Iterator Notes

There are many code examples available that demonstrate use of iterator classes. These examples also implement the functions necessary to generate and fill views and use predicates. Please come and talk with us about designing and implementing the iterators shown in the diagram.

6 Other Notes

All the Classdef macros from ROOT need to be removed from the code.