PHENIX Technical Note

PHENIX Track Models and Object-Oriented Hit Association

by Jeffery T. Mitchell
12/21/00

Abstract

This note will describe the general design of the object-oriented PHENIX central arm global tracking software. The implementation for the creation of DSTs from the data taken during the summer 2000 RHIC run will be described. This note addresses the general implementation strategies used and will not go into a description of specific track models or their performance. The note includes and concludes with simple suggestions to improve the hit association algorithms and reduce the need for DST after-burners.

I. Introduction

Global hit associations, defined as any association between hits or tracks of different detectors to a single particle track, are performed in the PHENIX central arm using an object-oriented design implemented with C++ within the PHOOL framework. All software is written independently of ROOT. This document will describe the details of this design for analysis of the summer 2000 run data. Described in this document are the motivation of the design, the details of the specific classes that have been developed, and how it all works together. The performance of specific implementations will be discussed in separate PHENIX notes. For more information and updates, please refer to the Luxor web pages at *http://www.phenix.bnl.gov/WWW/software/luxor/*.

When approaching the design of an object-oriented system, care must be taken in defining objects that are useful and efficient for the solution of the problem. For the problem of PHENIX global associations, the primary objects that are used are the following:

1. A track model object that describes the shape of a track in space.
2. Geometry objects that describe the positioning of the detectors in space.
3. A "*hit list*" object that stores the hits from a given detector that are available for association to a track.
4. Data Summary Tape *(DST)* objects that store the final association and kinematics results.

PHENIX global reconstruction anchors itself upon these objects, with classes built around them containing methods that manipulate the objects to solve the global hit association problem, which is a daunting one in PHENIX central collisions. Each of these objects and the methods that use them will be described in subsequent sections.

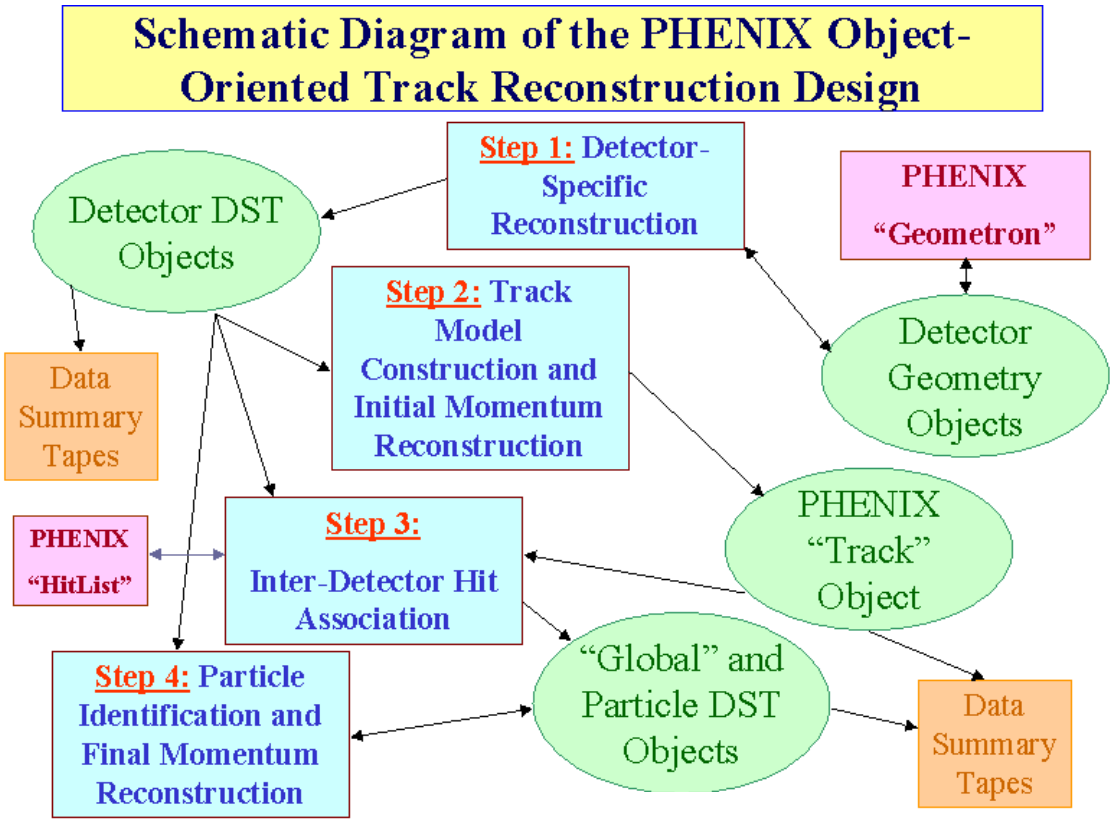

Figure 1: A schematic diagram showing the relationships between global reconstruction procedures and PHENIX reconstruction objects.

The procedure for performing PHENIX global hit associations is illustrated in Figure 1. The first step is to perform detector-specific reconstruction, whose output are PHOOL-based "*wrapped STAF*" DST tables. For this discussion, these tables can be considered as objects (they have data members and methods to access them) which will be referred to as *"detector DST objects"*. The specific DST objects in this category include vertex and centrality information from the BBC, MVD, and ZDC; drift chamber hits and tracks; pad chamber clusters; RICH PMT information; TEC tracks; TOF hits; and EMCAL clusters.

Once detector-specific reconstruction is completed, a track model class can use this information to construct PHENIX track model objects, referred to as *PHTracks*. A specific track model class can construct its track from any subset of the detector DST objects. For example, a track model can be based on drift chamber tracks as well as on outer detector information such as TEC tracks. No matter how a *PHTrack* object is constructed, the remainder of the global reconstruction operates in the same manner. One advantage of most track models is that they can obtain a prediction of the momentum of the particle, which is initially done assuming that the particle is a pion. All track models are required to perform projections to the other detectors, which are stored within the *PHTrack* object prior to global hit association, by using the PHENIX Geometry objects.

Once all *PHTrack* objects are created and their projections determined, the global hit association class can process them. To facilitate this, a "*hit list*" class, called

cglHitList, has been developed to help gather hit and track information and sort them according to arm, phi angle, or z coordinate. The output of the hit association is currently the wrapped STAF tables *dCglTrack*, which contains the association pointers, and *dCglParticle*, which contains the initial kinematics and particle ID information.

Following global hit association, it is then possible to perform a particle identification pass followed by a second momentum reconstruction pass on tracks that may not be identified as pions. These steps will not be discussed further in this note.

II. Geometry Objects

IIa. The geo package

One of the most important aspects of any global reconstruction algorithm is the handling of the geometry of the detectors being associated. In order to facilitate this, the PHENIX geometry objects were designed. Initially, the possibility of using the CLHEP geometry classes for our purposes was investigated, however not all of the necessary functionality was available. Given that fact along with concerns about the support of the CLHEP package on the short time-scale over which we needed its functionality prior to data taking, it was decided to write a geometry package specifically for PHENIX, called the *geo* package. Even though the *geo* package is PHENIX-based, it is written generally enough for any purpose. All geometry in the global hit association is defined in terms of the PHENIX geometry objects, which include points (*PHPoint*), cylinders (*PHCylinder*) and cylindrical sections (*PHCylinderSection*), vectors (*PHVector*), angles (*PHAngle*), reference frames (*PHFrame*), line segments (*PHLine*), planes (*PHPlane*) and planar sections (*PHPanel*), spheres (*PHSphere*) and spherical sections (*PHSphereSection*), lists of coordinates called poly-lines (*PHPolyLine*), and rotation matrices (*PHMatrix*). Manipulations upon a single geometry object by itself are included in that objects methods. However, methods that take input from more than one type of geometry object are included in a singleton class called the *PHGeometron*. This includes operations such as calculating the intersection point of a line and a plane.

Since the PHENIX geometry objects are C++ objects, it is possible to store them directly into the Objectivity database. The advantage here is that the reconstruction software can read these objects directly from the database. Alignment software could read a default geometry from the database, determine the best geometrical alignment, and then update the objects directly in the database for immediate use by the reconstruction software. At the time of this writing, a complete implementation of this is still under construction, yet well underway. It would be highly desirable to have a direct connection between the geometry objects stored in the database and the geometry used by the simulation, however work in that direction has not yet started.

More information on the geometry package is available on the web at *http://www.phenix.bnl.gov/WWW/software/luxor/geometry.html*.

IIb. The cglDetectorGeo class

A class, called *cglDetectorGeo*, is used in the reconstruction chain as a librarian, or clearing house, for detector geometry information. It serves to gather the geometry

information into a single storage place for use by the track models and global hit association. The class contains methods that extract the geometry from information stored in the database after alignment corrections have been applied (preferred). However, methods are also supplied to generate a default geometry matching the GEANT geometry (note that this match is not coupled to GEANT in any way, but can be controlled by run-time input parameters at the macro level). The *cglDetectorGeo* class can be defined as an object containing the necessary geometry that is stored in the PHOOL node tree.

The data members of *cglDetectorGeo* are designed to define the entire geometry needed for track models and hit association. The data members are described below:

- *xyz0*: A *PHPoint* object defining the PHENIX origin reference coordinate.
- *xyz0East*: A *PHPoint* object defining the PHENIX east arm origin reference coordinate. coordinate.
- xyz0West: A *PHPoint* object defining the PHENIX west arm origin reference *coordinate*.
- *phenixFrame*: A *PHFrame* object that defines the PHENIX reference coordinate system.
- *eastFrame*: A *PHFrame* object that defines the east arm reference coordinate system.
- *westFrame*: A *PHFrame* object that defines the west arm reference coordinate system.

The following data members are 2-D arrays indexed by arm and sector number:

- *pc1Sectors*: Array of *PHPanel* objects that define the PC1 sectors in space for each arm.
- *pc2Sectors*: Array of *PHPanel* objects that define the PC2 sectors in space for each arm.
- *pc3Sectors*: Array of *PHPanel* objects that define the PC3 sectors in space for each arm.
- *tecSectors*: Array of *PHPanel* objects that define the TEC sectors in space for each arm.
- *tofSectors*: Array of *PHPanel* objects that define the TOF sectors in space for each arm.
- *pbscSectors*: Array of *PHPanel* objects that define the PbSc sectors in space for each arm.
- *pbglSectors*: Array of *PHPanel* objects that define the PbGl sectors in space for each arm.
- *pc1Frame*: Array of *PHFrame* objects defining the reference coordinate system for each PC1 sector in each arm.
- *pc2Frame*: Array of *PHFrame* objects defining the reference coordinate system for each PC2 sector in each arm.
- *pc3Frame*: Array of *PHFrame* objects defining the reference coordinate system for each PC3 sector in each arm.

- *tecFrame*: Array of *PHFrame* objects defining the reference coordinate system for each TEC sector in each arm.
- *tofSectorsFrame*: Array of *PHFrame* objects defining the reference coordinate system for each TOF sector in each arm.
- *tofPanels*: Array of *PHFrame* object defining the reference coordinate system for each TOF panel in each arm.
- *pbscFrame*: Array of *PHFrame* object defining the reference coordinate system for each PbSc sector in each arm.
- *pbglFrame*: Array of *PHFrame* objects defining the reference coordinate system for each PbGl sector in each arm.

These data members are 1-D arrays indexed by arm number:

- *dchArm*: PHCylinderSection objects that define the drift chamber geometry for each arm.
- *crkArm*: PHCylinderSection objects that define the RICH geometry for each arm.
- *dchFrame*: PHFrame object defining the reference coordinate system for each drift chamber arm.
- *crkFrame*: PHFrame object defining the reference coordinate system for each RICH arm.

These data members point to subsystem geometry objects for subsystems that are handling their geometry according to PHENIX specifications:

- *PHpadDetGeo*: The pad chamber detector geometry definition object.
- *TofDetGeo*: The TOF detector geometry object.

III. The PHENIX Track Object

The track object, called a *PHTrack*, represents the shape of the track in space. The shape of a track is determined by a track model, which is implemented as a class that inherits from the *PHTrack* base class. The job of a track model is to reconstruct one *PHTrack* object for each particle track in the spectrometer. The track model can use whatever information is available from the detectors to construct *PHTracks*. The track model must also provide methods which intersect the track with each detector, and methods which define the shape of the track in terms of a *PHPolyline* geometry object for event display.

Below is a more detailed description of the PHTrack data members:

- *polyline*: This is a *PHPolyLine* geometry object (a list of 3-D points in space) which describes the shape of the track in space.
- *trackIndex*: This is a pointer to the *dCglTrack* object which stores the association information. This is usually filled by the global hit assocation algorithm rather than the track model.
- *arm*: The arm number (0 = East arm, 1 = West arm)

- *ifIntersect*: This is a *PHPointerList* of *PHBoolean* quantities that specify whether or not the track model can intersect the track object with a given detector. This list is always the length of the number of detectors that are in the intersection list, and the quantity is set if there was a successful intersection. The ordering of this list is always the following: 0 = Vertex (closest approach to the vertex in the x-y plane), 1 = drift chamber, 2 = PC1, 3 = PC2, 4 = PC3, 5 = RICH, 6 = TEC, 7 = TOF, 8 = PbSc, 9 = PgGl.
- *projections*: A *PHPolyLine* object which contains the projection coordinates of the track at a given subsystem. This only has entries in the list if there was an intersection. However, the list is filled in the same order as the *ifIntersect* list. If *ifIntersect* for a given detector is false, then the *projections* list is not filled for that detector. In order to determine which detector the projection corresponds to, you must first query the *ifIntersect* list.
- *projErrors*: A *PHPolyLine* object that contains the estimated errors of the projections at each successful intersection point of the track object with the detectors. These errors are 1 standard deviation errors in each Cartesian direction. The estimation of these errors are used by the global hit association to determine the weight of an intersection (or a road width) for the association. In this way, momentum-dependent weights can be applied. The ordering of the errors in *projErrors* is identical to that of the *projections* object.
- *directions*: A *PHPointerList* of *PHVector* objects which contain the track vector at each successful intersection point of the track object with the detectors. The ordering of the list is identical to that in the *projections* data member.

Below is a detailed description of the *PHTrack* virtual methods (that is, the track model should supply these methods, although the base class provides default methods for some functions):

Each *projectTo...* method returns a *PHBoolean* indicating whether or not there was an intersection. Intersections are only allowed to occur within the arm containing the track.

- *projectToVertex*: Distance-of-closest-approach projection to the z-axis.
- *projectToDch*: Projection to the reference cylinder of the drift chamber.
- *projectToPc1*, *projectToPc2*, *projectToPc3*: Projection to the pad chambers.
- *projectToCrk*: Projection to the reference sphere of the RICH.
- *projectToTec*: Projection to the TEC.
- *projectToTof*: Projection to the TOF.
- *projectToPbSc*, *projectToPbGl*: Projection to the EMCAL.

The default algorithm for the next 3 methods is to intersect the *polyline* in *PHTrack* with the input geometry object.

- *projectToPlane*: Projection to an arbitrary *PHPlane*.
- *projectToCylinder*: Projection to an arbitrary *PHCylinder*.
- *projectToSphere*: Projection to an arbitrary *PHSphere*.

- *predictMomentum*: Most track models can return an estimate of the track momentum vector.

The default algorithm for the next 3 methods is to calculate the path length defined in the *polyline* in *PHTrack*.

- *pathLengthToCrk*, *pathLengthToTof*, *pathLengthToEmc*: Calculate the total path length from the collision vertex to the given detector.

- *calcPolyLine*: Fill the *polyline* object.

- *callProjections*: Calls all detector-based *projectTo* methods at once.

It is highly desirable to eventually write out *PHTrack* objects directly to the DSTs, although they are stored in the PHOOL node tree. This just needs a small amount of work for someone to test and possibly tweak the streamer methods for *PHTrack*. The technique used within the header package can be applied here to keep ROOT out of the compiled code by providing a buffer. However, in the meantime, a wrapped STAF object called *dPHTrack* is filled and stored in the DSTs instead. *dPHTrack* contains all of the information in *PHTrack* objects. Unfortunately, this information is currently and unnecessarily being stored twice in the DSTs, since it is also present in the *dPhDchTrack* object when the PHDchTrack track model is used.

IV. Hit List Objects

Much of the work to be performed in any hit association algorithm of the scope needed for the PHENIX central arms is concerned with hit and track bookkeeping. Many of the algorithms that must be applied can be sped up significantly if they operate on ordered sets of hits or tracks. Many of these generic bookkeeping functions have been gathered together in a single class for use within PHENIX hit association algorithms. The name of this class is *cglHitList*. Generally, a *cglHitList* object is simply a list of coordinates. The class provides specific constructors for many of the central arm detectors including drift chamber tracks, pad chamber clusters, TEC tracks, TOF hits, and EMCAL clusters. The constructors can be specified to create a hit list from one or both arms. Once a *cglHitList* object is created, the sorting and searching methods included in the class, which are described below, can be applied.

The data members of a *cglHitList* object are as follows:

- *n*: The number of entries in the list.
- *sortflag*: Specifies whether or not the list is ordered. This is set to 0 if there is no ordering, 1 if it is sorted in order of increasing phi, and 2 if it is sorted in order of increasing z.
- *detid*: The detector type identifier. The key is 0 = drift chamber, 1 = PC1, 2 = PC2, 3 = PC3, 4 = RICH, 5 = TEC, 6 = TOF, 7 = PbSc, 8 = PbGl, -1 = other.
- *arm*: The arm number for the hits. Set to -1 for no specification. Set to 2 for both arms. Set to 0 for the East arm, and 1 for the West arm.

- *index*: An array of pointers to the DST entry for that detector from which the coordinate information was obtained.
- *coord*: An array of *PHPoint* objects containing the coordinates for easy access.
- *Verbose*: Verbosity level for debugging. 0 is no output to the screen.

The methods provided with *cglHitList* include the following:

- *Print*: Print the entire list in Cartesian coordinates.
- *PrintCyl*: Print the entire list in Cylindrical coordinates.
- *SortInPhi*: Sort the list in order of increasing phi coordinate.
- *SortInZ*: Sort the list in order of increasing z coordinate.
- *Remove*: Remove a specific entry in the list.
- *Clear*: Remove all entries in the list.
- *Add*: Add a coordinate in a specific entry in the list.
- *Append*: Add a coordinate to the end of the list.
- *ZRange*: Return all entries that are within a specified z range.
- *PhiRange*: Return all entries that are within a specified phi range.
- *PhiZRange*: Return all entries that are within a specified z-phi window.
- *PhiClose*: Return the entry of the closest hit in phi to the input point within a specified z-phi window.
- *ZClose*: Return the entry of the closest hit in z to the input point within a specified z-phi window.
- *PhiZClose*: Return the entry of the closest hit to the center of a specified z-phi window in 2-D.

You might notice that this looks a lot like something that could be implemented with STL, however the time pressure for the completion of this class prior to PHENIX's first data precluded that possibility due to lack of training time.

V. Global Hit Association

This section describes the algorithm for the *cglHitAssociate* class used for DST production for the Run 2000 data.

The algorithm used in *cglHitAssociate* is very simple, with most of the work done by the track models and the *cglHitList* class. The entirety of the algorithm is contained within the *event* method, which operates on all *PHTrack* objects for an event in a single pass.

First, the PHOOL node tree is searched for the data that is necessary for hit association. If data is missing, the event method will terminate with an error. The output DST information in *dCglTrack*, *dCglParticle*, and *dPHTrack* are initialized for the event.

Each arm is looped over. Tracking is done on an arm-by-arm basis. For each arm, *cglHitList* objects are constructed for each detector. These lists are sorted in order of increasing phi after they are constructed.

Every reconstructed drift chamber track is looped over. Hit association is performed on a track-by-track basis. A pointer to the *PHTrack* object (keyed by track

model using the *TrackModelFlag* input parameter) is created. All projection information and track model methods are accessed via this pointer for the hit association to this track. It is assumed that there is a one-to-one correspondence between drift chamber tracks and *PHTrack* objects. If this assumption is ever broken, the code will need to be modified accordingly. Only drift chamber tracks within the current arm that pass the cut specified by the *dchQualityFlag* input parameter are considered for hit association.

Each detector is looped over using the following indexing scheme: 0=vertex, 1=dch, 2=PC1, 3=PC2, 4=PC3, 5=RICH, 6=TEC, 7=TOF, 8=PbSc, 9=PbGl. A detector is only considered for association if the input parameter *UseFlag* for that detector is set.

For each detector, the track model is queried for the projection point, projection error, and projection vector at that detector. Using the projection point and error, a window about the projection is formed based upon the input parameters *PhiWidth*, *ZWidth*, *MinPhiWidth*, and *MinZWidth*. The *cglHitList* object for that detector is then used to determine the closest hit within the window to the projection point. If there is a valid hit to associate, the pointer to its DST object is stored in *dCglTrack*. The z information may not be used to determine the closest hit if specified by the *PhiRoadOnly* and *MaxDchQuality* input parameters. When associated a TEC track, an added cut on the slope difference is applied, as specified by the *TECSlopeCut* input parameter.

Once all hit association is complete for a given input drift chamber track, the rest of the known quantities in *dCglParticle* and *dPHTrack* are filled. The momentum prediction from a track model is filled if the *PredictMomentum* input parameter is set.

There are some improvements that can be made to immediately increase the efficiency of this algorithm. They include the following:

- The *RemoveHits* functionality has not yet been tested. First, does it work? Second, does it improve performance (you can use the *cgeEvalTrack* class for a quick look)?
- Currently, only the closest hit for any given detector is associated to the drift chamber track. Every detector is treated completely independently. An optimization could easily be performed with little code modification as follows: 1. Use *cglHitList* to make a list of every hit for every detector within the windows. 2. Write a method(s) that fits or optimizes these collections of hits (e.g. a linear chi-square test on the outer detector hits) to find the best set for storage in *dCglTrack*. 3. Evaluate the performance using the *cglEvalTrack* class. A modification here may be more desirable considering CPU time and ease-of-use than the proliferation of DST after-burners seen in PHENIX analysis software.
- Currently, only one set of hits is stored for every drift chamber track. It may be desirable for some analyses to make these decisions later (maybe after some PID comparisons are applied). Install a redundancy depth to store a maximum number of hit sets (maybe ordered in confidence) to the DST information. Be careful though, some analyses may be assuming the current one-to-one correspondence between drift chamber and global tracks. Make sure you let everyone know well in advance that you are installing this feature. Again, installation of this functionality here may be more efficient and easier to use than a DST after-burner.

- The RICH software has been implemented in a way that ignores CPU-time and manpower efficiency arguments and includes independently developed global associations. Due to this implementation, many procedures are being repeated (and rewritten) within a single PHENIX reconstruction chain. Uniformity in this regard may be highly beneficial to PHENIX data analyses.
- Most track models are currently making no attempt to return errors at the detector projection points. Tracking model authors should note that this is a desirable output so that momentum-dependent windows, rather than the default flat windows, can be defined for hit association. If the errors appear, implementing this is automatic - no need to even change the input paramaters or the software. This could easily increase the association efficiency.
- The task of optimizing the input parameters to *cglHitAssociate* to maximize efficiency has not yet been tackled. With very little work, the performance of this class could be increased greatly.