CDF/DOC/COMP_UPG/CDFR/5870
Version 4.8.4
November 4, 2002

A Guide to Muon Reconstruction and Software for Run 2

James N. Bellinger, Ken Bloom, W. David Dagenhart, Andreas Korn,
Slava Krutelyov, Victoria Martin, Michael Schmitt

Abstract

We describe the CDF muon-reconstruction software for Run 2 – the algorithms, the output data structures, the overall framework, and auxillary tools.

CDF's muon-detector systems have undergone many changes in preparation for Run 2[1]! Some chamber and scintillator systems are entirely new, while the old systems have refurbished electronics and data-acquisition systems. At the same time, CDF's offline system has changed markedly. The operating framework is now written in C++, and virtually all of the software has been written or re-written in that language. The use of ROOT as the disk-resident data format has led to huge changes in the event-data model and the data structures that are used to carry the output of event reconstruction – the basis of the physics that we all aspire to study.

These changes have led us to undertake what is effectively a complete rewrite of the muon-reconstruction software for Run 2. In doing so, we have tried to make the best use of the features of C++ and the new data format, while trying to avoid the pitfalls and to keep the software straightforward, so that the average user can understand the reconstruction process. But our top priority has been to write software that reconstructs muons efficiently, with little impurity, with the minimum amount of processing time, so that we can get at the best physics as quickly as possible.

In this note, we describe the muon-reconstruction software. We begin with a simple overview of the organization of the software, and how to run it, and of the data structures that are the output of the software – the hits, stubs, and "muons." We provide some technical details on matters of geometry and calibration, and then turn to a description of the reconstruction algorithms. The performance of the reconstruction is, for the most part, discussed elsewhere [2].

1 Running the Muon Reconstruction

As in the rest of the offline system, the muon-reconstruction code is broken into several `AC++` modules which are in the `MuonMods` package. The modules come in three types, which are run in succession. The first are known as "D to E" modules, an entirely historical term from Run 1. These modules take raw data from the detector (D banks) and apply calibration constants and do some noise suppression. There is one D to E module per muon subsystem, and there are eight muon subsystems. (These are the four chamber systems – CMU, CMP, CMX, BMU – and the four scintillator systems – CSP, CSX, BSU, TSU). Each module has the name `XXX_DtoEModule`, where `XXX` is the three-letter acronym of the subsystem that produces the data processed by the module.

The second type of module performs pattern recognition on the hits and finds stubs in the chamber systems. There are three of these – the `MuonStubModule` which acts on the CMU and CMP data, the `CMX_EtoSModule` that acts on CMX data, and the `BMU_StubModule`, which acts on BMU data. ("E to S" is another historical term.)

The last type of module takes stub data and track data and forms associations between tracks and stubs, *i.e.* muons. There is only one module of this type, the `MuonLinkerModule`.

Thus, there are a total of 12 modules that must be run, with a non-obvious series of names. To simplify life for the user, these modules are packaged together in the `MuonRecoSequence`, also in the `MuonMods` package. This sequence can be included in your `AppUserBuild` class, just as any single module is included. Then, the `MuonRecoSequence` can be added to a path, just like any individual module can. Each module has a number of talk-to options that can be set, and most of these will be discussed in the section on algorithms.

By default, the modules assume that the muon reconstruction is being done "fresh," starting from the D banks, and ignoring any hits, stubs, or tracks that might already be in the input file. This is the most likely scenario, as the muon reconstruction will typically run in `ProductionExe` on a raw-data file or on `cdfSim` output. You may wish to run just part of the reconstruction using already-existing structures as input, *e.g.* you may want to run only the `MuonLinkerModule` using stubs and tracks in the input file. Each module has talk-to parameters that allow you to do this. By setting `findProcessName` to `true`, the module will use the hits (if it is a stub-finding module) or stubs (if it is the linker module) that were created with a given process name, specified with the `processName` talk-to parameter, rather than those that were created during the running of this same program. In the `MuonLinkerModule`, `findTrackProcessName` and `trackProcessName` are the corresponding talk-to parameters to specify the use of an existing set of tracks. Note that there are two steps required in all of these modifications – just changing the process name will not have any effect unless the appropriate `find` talk-to is enabled. A sample script to completely redo the muon reconstruction on your own is given in Appendix A.

2 Data Structures

Once you have run the muon-reconstruction modules, you will want to examine their output. There are different data structures which hold the output of each of the three types of modules. The D to E modules create `StorableObjects` of types `MuonXZHitColl`, `MuonXHitColl`, and `MuonTHitColl`. These contain objects of type `MuonXZHit`, `MuonXHit`, or `MuonTHit`, accordingly. The stub modules produce `StorableObjects` of type `MuonStubColl`, which contain `MuonStub` objects. Finally, the `MuonLinkerModule` creates a `CdfMuonColl` containing `CdfMuon` objects. Below, we discuss the basic features of each of these data structures.

2.1 Hits

The different subsystems of the muon detector produce different kinds of information. The four chamber systems measures a drift time, which at D to E time is converted to a drift distance, *i.e.* a distance from the wire to a location that the muon had occupied in its flight, in the plane perpendicular to the chamber sense wire . However, the CMU and the BMU have means to measure the position of the muon along the sense wire, which the CMP and CMX do not have. Thus, some chambers produce data leading to one distance measurement (the drift distance), while others produce data leading to two such measurements (the drift distance and the "z" distance). On the other hand, the counter systems only produce timing information, which cannot be turned into position information in any straightforward way.

Therefore, we have developed three kinds of data structures for muon hits. The `MuonXZHit` is the output of the CMU and BMU D to E modules, holding two distance values. The `MuonXHit` is the output of the CMP and CMX D to E modules, holding a single distance value. The `MuonTHit` is the output of each scintilltor D to E module, holding a time.

However, each of the three kinds of hits also carry some common information. These include the geometric address of the chamber or scintillator that produced the hit, which is known as a `MuonDigiCode` and described in greater detail in the section on geometry; the raw TDC information that is the basis for the hit; a noise flag; a flag indicating that the z information in the hit is valid; and, for simulated data, the `OBSP` number of the particle that produced the hit. To handle this commonality, the three hit classes are children of the `MuonHit` abstract class, giving them a uniform interface. Among the accessors of all `MuonHits` are:

- `MuonDigiCode getCode()` – returns the `MuonDigiCode` for the chamber or scintillator that produced the hit
- `HepPoint3D getLocalPosition()` – returns the position of the hit in "local" coordinates
- `bool isNoise()` – `true` indicates that the hit has been flagged as a noise hit
- `bool hasValidZ()` – `true` indicates that the hit has been valid z information
- `int getMcId()` – returns the OBSP number of the creating particle, if available, or -1 if not; only one OBSP number can be returned, even if more than one particle contributed to the hit
- `unsigned int getTDCTime(), getTDCWidth()` – return the raw TDC time and width that are the basis of this hit

The three subclasses have several methods to themselves. The `MuonXZHit` and `MuonXHit` both have a `getDriftDistance` method, which returns the drift distance as calculated at D to E time. The `MuonXZHit` also has a `getZDistance` method, which returns the calculated position along the wire, in local coordinates. The `MuonTHit` has a `getTime` method that returns the scintillator TDC time after correction by calibration constants.

The idea of "local coordinates" is discussed in the section on geometry, but suffice it to say here that the `getLocalPosition` method returns `(getDriftDistance(), 0., getZPosition())` for a `MuonXZHit`, `(getDriftDistance(), 0., 0.)` for a `MuonXHit`, and `(0., 0., 0.)` for a `MuonTHit`.

Note that the `MuonHit` data structures contain *only* information that can be determined at D to E time from individual hits. These hits do not contain drift signs, since these cannot be determined without a pattern-recognition algorithm running on multiple hits. Drift distances are therefore always greater than zero. They do not take into account particle entrance-angle corrections, which also require pattern recognition.

Without a drift sign, it is impossible to determine a true position of a `MuonHit`; there is a two-fold ambiguity. Thus, there is a `MuonSignedHit` class, which holds a pointer to a `MuonHit` and a drift sign. The `MuonSignedHit` class has a `getGlobalPosition` method, returning a `HepPoint3D` in global (CDF) coordinates, calculated using information from the geometry description, the local coordinate, and the drift sign. This point should be taken only as an approximation of the location of the hit. A better estimate of the hit location requires a drift model and a particle entrance angle, as will be discussed below. The `MuonSignedHit` has a `getStubResidual` method that will give a residual with respect to an input `MuonStub`; this is also discussed in greater detail below.

2.2 Hit Collections

Each muon subsystem can produce many `MuonHits`, so they are collected into `MuonHitColls`, which are `StorableObjects`, and thus can be stored in the event record. There is a different `MuonHitColl` for each kind of MuonHit, *i.e.* MuonXZHitColl, MuonXHitColl, and MuonTHitColl. Each of these three classes inherits from MuonHitColl, and thus they all have the same interface.

When any MuonHitColl is pulled out of the event record, it can then be accessed either as a generic MuonHitColl, or as one of the three specific hit collections. If the collection is treated as the generic type, then only the generic MuonHits are accessible, while the specific collection allows access to the specific hit types (XZ, X, or T). This is probably the trickiest concept in understanding how the hit collections work.

The average user is most likely to use some methods of the MuonHitColl that put hits into simple STL vectors, which you are probably more familiar with than the hit-collection structure. Some of these functions are:

- getAllHits(vector_type& hits) – fills hits with all of the hits that are in the hit collection
- getAllSignedHits(MuonSignedHitVector& hits) – creates two MuonSignedHits for each hit in the collection (one of each drift sign), and fills hits with them
- getSignedHitsInLayer(MuonDigiCode code, MuonSignedHitVector& hits) – fills hits with MuonSignedHits for channels in the layer specified in code
- getSignedHitsInRegionLayer(MuonDigiCode code, MuonSignedHitVector& hits) – fills hits with MuonSignedHits for channels in the region and layer specified in code

Here, vector_type is defined as std::vector<HIT>, where HIT is one of the MuonHit classes, and MuonSignedHitVector is defined as std::vector<MuonSignedHit>. All of these methods take additional optional arguments that allow you to specify that only hits that are not marked as noise, or hits that are on channels that have fewer than some maximum number of hits, are included in the vectors.

Several features are more easily used by users with a stronger command of C++. For instance, the specific hit collections allow access to the underlying container of hits through the contents method. This container is a MapVector, which is defined in the STLExtensions package, and is effectively a "vector of vectors." Each of the hits on a given channel in the system are put into a STL vector. Then, these vectors are collected together in the MapVector, and are indexed by MuonDigiCode. Thus, an iteration over all hits in the MapVector requires a double iteration – an iteration over all the vectors in the MapVector, and then an iteration over each hit in a given vector. This data structure also allows the easy retrieval of all the hits on a given channel, since it is (relatively) straightforward to find the vector associated with a given MuonDigiCode through the find method.

Another "expert-level" feature of the hit collections is that individual MuonHits (either generic or specific, as appropriate) can be accessed directly from the collection with an index value. The hit collection can calculate these from a pair of iterators (which were described above). Such an index is hidden away inside the MuonSignedHit. As we shall see, the stub object holds a collection of MuonSignedHits, and the stub uses the indices to locate the hits in their parent hit collections when an event is read in from disk to memory.

2.3 Stubs and Stub Collections

The output data structure of the stub-finding algorithms is the streamable MuonStub object; these are collected into MuonStubColls that are StorableObjects and can be placed in the event record.

A MuonStub holds the hits on the stub, and the fit parameters for the stub. The hits are stored internally in a std::list<MuonSignedHit>. In the code, the signed-hit class is aliased as

AssHit, which means "associated hit." The fit outputs are set of parameters, the errors on those parameters, and χ^2 values for the $x-y$ and $x-z$ parts of the fit. The output of the fit is a stub position and a stub direction; the direction is a unit vector that points from the stub position point in the direction of the muon's flight, so the position and direction together define a line segment.

In addition, the MuonStub holds counter hits that are associated with the chamber stub – CSP hits for the CMP, CSX hits for the CMX, and TSU and BSU hits for the BMU. They are stored internally in a std::list<MuonTHit>, and MuonTHit is aliased as CounterHit. In principle, a stub could have several combinations of hit information (chamber and counter hits, only chamber hits, only counter hits) and could arise from several different algorithms (add counter hits to a chamber stub, use counter hits to seed the chamber-stub finding, *etc.*), so the MuonStub has quality and algorithm values to indicate this information. There are also separate t_0 values for the chamber and counter hits, should they be extracted from a fit.

The accessors of the MuonStub object include (but are not limited to):

- HepPoint3D spacePoint() – position of the stub
- HepVector3D direction() – direction of the stub
- float chiSquaredX(), float chiSquared() – χ^2 of the $x-y$ fit
- float chiSquaredZ() – χ^2 of the $x-z$ fit
- numAssHits() – number of chamber hits on the stub
- AssHitCIter beginAssHits(), endAssHits() – iterators pointing to beginning and end of the list of associated chamber hits; dereferences to a MuonSignedHit
- numCounterHits() – number of counter hits on the stub
- CounterHitCIter beginCounterHits(), endCounterHits() – iterators pointing to beginning and end of the list of associated counter hits; dereferences to a MuonTHit

Other methods return the errors on the fit parameters, a pointer to the MuonHitColl that contains the MuonHits which are the basis for the stub, a fit-status word, and the OBSP number of the associated simulation particle, which is determined from the OBSP numbers of the associated hits.

The MuonStubColl is a StorableObject that holds a vector of MuonStubs, which is returned by the contents method. This vector can be iterated on in the usual way. The stubs for each chamber system are placed in a separate MuonStubColl in the event record, which are distinguished by a description string.

2.4 CdfMuon

Reconstructed muons are represented by the CdfMuon object. It holds pointers to the track and the stub or stubs that form the muon, and information about the quality of each track-stub match. It also includes information about energy deposition and calorimeter-isolation quantities for the associated track. Every CdfMuon is guaranteed to be associated with a CdfTrack. It may or may not have a stub in any given muon subsystem. The track-stub matching quality information is kept in a MuonMatchData object; there is one of these for each associated stub.

It is also possible to have a CdfMuon with no stubs attached at all (and therefore no track-stub matching information). If these "stubless muons" are minimum-ionizing and do not extrapolate to the fiducial volume of the muon system, then some analyses might want to use these objects as muons. This kind of `CdfMuon` does have a pointer to the track and the calorimeter information is filled.

Accessors that return information about tracks include:

- `CdfTrack_clnk bestTrack()` – a link to the primary track associated with the muon; the track itself can be obtained *via* `*(muon.bestTrack())`
- `HepLorentzVector fourMomentum()` – the four-momentum of the muon, as determined from `bestTrack()` and the muon mass
- `CdfTrack_clnk secondBestTrack()` – the next-best track for the stub(s), as determined by the track-stub linking algorithm

A muon will not necessarily have a next-best track, so the `secondBestTrack` method may return a null link; this should be tested before the link is used.

Accessors that return information about stubs include:

- `int muonType()` – a bitword indicating the muon subsystems that contributed stubs to this system; see `MuonDigiCode` for the bit assignments
- `bool hasCmu(), hasCmp(), hasCmx(), hasBmu()` – indicates whether a given subsystem has contributed a stub
- `vector<const MuonStub*> muonStubs()` – returns a vector of pointers to all of the `MuonStub` objects associated with this muon that are in the event record
- `MuonStub* stub(unsigned int system)` – returns a pointer to the `MuonStub` in `system` associated with the muon; pointer is `NULL` if there is no stub in that system

The user should be careful about trying to use a pointer to a `MuonStub` for a system where there is no stub; it is best to test the pointer, or to check on the return value of the `has` methods.

Accessors that return information about the track-stub match quality include:

- `MuonMatchData& matchData(unsigned int system)` – returns the `MuonMatchData` for the stub in the given system; will return an illegal value if there is no stub in that system
- `MuonMatchData& cmu(), cmp(), cmx(), bmu()` – returns the `MuonMatchData` for the stub in the given system; will return an illegal value if there is no stub in that system
- `HepPoint3D trackPosition(unsigned int system), HepVector3D trackDirection(unsigned int system)` – return the track position and direction in the given system, for comparison with the stub position and direction; will retun an illegal value if there is no stub in that system

Again, the user should be careful about trying to use a `MuonMatchData` object for a system where there is no stub.

The `MuonMatchData` object has these accessors:

- `float drphi()` – $r\Delta\phi$ between the position of the stub and the track extrapolation to the stub, in centimeters
- `float dx()` – same as `drphi()`, common CDF jargon

- `float dphi()` – the opening angle between the stub direction and the track extrapolation to the stub, projected onto the $r\phi$ plane, in radians
- `float dz()` – the difference in z between the position of the stub and the track extrapolation to the stub, in centimeters
- `float dtheta()` – the opening angle between the stub direction and the track extrapolation to the stub, projected onto the rz plane, in radians
- `chisqX()` – the χ^2 value representing the quality of the track-stub match, formed from the `dx` and `dphi` values
- `chisqXPosition()` – the χ^2 value representing the quality of the track-stub match, formed from the `dx` value only
- `chisqZ()` – the χ^2 value representing the quality of the track-stub match, formed from the `dz` and `dtheta` values
- `chisqZPosition()` – the χ^2 value representing the quality of the track-stub match, formed from the `dz` value only

The quantities above are all in the CDF global coordinate system. At the moment, most of the χ^2 quantities are not filled; only `chisqXPosition` and `chisqZPosition` for the CMU return useful values. The other quantities can be filled by the `GEANT`-based extrapolator, which is not available for use in the production executable. Instructions for re-making muons this way are given in Appendix B.

Finally, among the `CdfMuon` accessors that return calorimeter quantities are:

- `float emEnergy()` – the energy deposited by the track in the electromagnetic calorimeter
- `float hadronEnergy()` – the energy deposited by the track in the hadronic calorimeter
- `const TowerKeySet& hadTowers(), emTowers()` – returns a `std::set<TowerKey>` that identify the calorimeter towers traversed by the track

Here, "the track" is defined as the primary track associated with the muon. There are also accessors for various isolation sums.

Starting from the `CdfMuon`, a user can access information about every object associated with the muon. The `CdfMuon` itself provides direct access to the track (through a link to the `CdfTrack`), and to all the stubs by returning `MuonStub` pointers. From the `MuonStub`, the user has direct access to all the `MuonSignedHits` associated with the stub, and then from those hits there is access to the drift distances, and even to the original TDC information that were the basis for these hits. These linkages between the muon, stub and track objects are preserved when the objects are written to a file, and restored when the objects are read back from the file into memory. It is possible (but not advised) to delete the hit collections and the stub collections from the event record. The basic information about the muon, including the matching quantities, will be kept, but obviously links to the stubs and hits will be lost.

2.5 CdfMuonColl and CdfMuonView

The CdfMuon object itself is not stored directly in the event record. The StorableObject is the CdfMuonColl, which holds a vector of CdfMuons. As with the MuonStubColl, this vector can be accessed through the contents method. The output of the MuonLinkerModule is a CdfMuonColl which has the description ProductionMuons.

However, the use of the CdfMuonView is encouraged for accessing CdfMuons. Views are collections of pointers to muons, and they allow for easy pre-selection and sorting of the muons. By contrast, the CdfMuonColl is not sortable, and individual muons cannot be easily excluded. The CdfMuonView also has a contents method that returns a vector of pointers to CdfMuons. An iterator over this vector must be dereferenced twice to get a CdfMuon – once to dereference the iterator to a pointer, and once to dereference the pointer.

An easy way to get a CdfMuonView is through the static method bool defMuons(CdfMuonView_h handle), which fills handle with a CdfMuonView constructed in a "smart" way. This method will try to return the muons in a CdfMuonColl with the description ProductionMuons that was created in the executable currently running. If it fails to find that, it will fall back to one that was created by ProductionExe, and then to one created by Level 3. The defMuons function can also take a StorableObject::Selector argument, which will put the user's choice of restrictions on the CdfMuonColl objects with the description ProductionMuons that are selected. The defMuons method will only return CdfMuons which have stubs associated with them. A companion function, defStublessMuons, will fill a view with CdfMuons that do not have stubs associated with them, as described in the section on track-stub linking. There is also a static method called allMuons, which will gather all of the muons in all of the CdfMuonColls in the event; this blunt instrument is not recommended.

Once a CdfMuonView has been obtained, it can filtered according to a selection cut. A new CdfMuonView can be constructed:

```
CdfMuonView newView(oldView.contents().begin(),oldView.contents().end(),
                    selector);
```

where selector is one of the "selector" classes defined in MuonObjects/muon_cut.hh. These selectors allow for selection by muon system, or by cuts on a track. The contents of the views can be sorted *via*

```
std::sort(view.contents().begin(), view.contents().end(), comp);
```

where comp is one of the comparison classes defined in MuonObjects/muon_sort.hh. At the moment, there are very few selector and comparison classes, but more can be added if there is sufficient demand.

3 Geometry

3.1 Numbering Conventions and MuonDigiCode

Each muon subsystem has its own unique geometry, due to the physical constraints of the size and shape of the collision hall. As a result, different numbering schemes have sprouted for the elements of the each system. We have tried to expose some uniformity across the subsystems through the use of the MuonDigiCode object as a channel ID. Every MuonHit carries a MuonDigiCode, to identify the chamber or counter that the hit belongs to.

Fortunately, almost all of the subsystems can be broken down into similar units. They are typically divided into first into "sides" (east or west) and then into "wedges," which is a natural physical geometry for the CMU, and a geometry that is created for the CMX, BMU, CSX, BSU and TSU by the data-acquisition system. Each wedge is then divided into "stacks," which are sets of chambers at approximately the same ϕ in CDF coordinates, but differing distances from the interaction point. Stacks are then broken down into layers. Thus, for most systems, a given chamber or counter can be identified by system, side, wedge, stack and layer numbers.

This does not quite hold in the CMP and CSP, where there is no natural breakdown into wedges. Instead, each of the four "regions" (north, top, south and bottom) is treated as if it were a single "wedge," with the stack and layer numbers assigned to each chamber or counter restarting from zero in each region. The CMP only has one "side," as single chambers cover the entire z range.

Thus `MuonDigiCode` has these accessors, among others:

- `unsigned int getSystem()` – returns the system code
- `unsigned int getSide()` – returns the side number
- `unsigned int getWedge(), getRegion()` – returns the wedge or (for CMP and CSP) region number
- `unsigned int getStack()` – returns the stack number
- `unsigned int getLayer()` – returns the layer number

The system code is defined as CMU = 0, CMP = 1, CMX = 2, BMU = 3, CSP = 4, CSX = 5, BSU = 6 and TSU = 7. The side number is 0 for west side and 1 for the east side. The region numbers for the CMP and CSP are north = 0, top = 1, south = 2, and bottom = 3. The `MuonDigiCode` class has `enum` values defined, so that the user does not need to remember these magic numbers.

In general, these numbering schemes follow those for the data acquisition system as specified in CDF 4152 [3]. The numbering of wedges starts at zero from $\phi = 0$, and stack numbering starts over from zero with the stack at the smallest ϕ within each wedge. Layer numbering starts at zero with the layer that is closest to the interaction point. This convention holds even in the CMP and CSP; the CMP north region is assigned number 0 since it covers $\phi = 0$ and the chambers with the smallest stack numbers are at approximately -45 degrees in ϕ. This stack numbering for the CMP and CSP are not followed in the data-acquisition system. Some muon subsystems have special `MuonDigiCode` accessors, to give numbering schemes more reflective of their geometries. In particular, the pattern recognition algorithms for the CMX and IMU systems tend not to break their detectors down into wedges, and use a continuous ϕ index instead.

3.2 Detector Node and Moving Parts

As with all CDF geometry, the description of the muon-system geometry is handled through a class that inherits from `CdfDetectorNode` – in our case `MuonDetectorNode`. When this singleton object is built during the begin-job phase of the `GeometryManager` module, a collection of objects inheriting from `CdfTangible` is built. Each of these objects represents a unique active detector element; each one knows its dimensions, its material, and its location in space. The last of these is defined by a "transform" that, when applied to a `HepPoint3D` in local coordinates of that particular tangible, gives the global coordinates of that point.

All sorts of conventions in the muon-geometry description are woefully inconsistent, and are often confusing even to experts, so we do not attempt to describe them in this note for the general public. But suffice it to say that the `MuonDetectorNode` contains a series of maps that allow the user to find various geometry objects, and their transforms, by specifying the `MuonDigiCode` of a particular tangible.

Unlike most of the rest of CDF, the location of many of the muon detectors can be changed in the collision hall. The north and south CMP walls can be retracted by as much as a meter, and the CMX and IMU (along with the toroids) can be moved along the beamline. It has not been unusual for CDF to run with these detectors out of their nominal positions. This can be accounted for by repositioning the tangibles in the muon-geometry description. At begin-run time, the `MuonDetectorNode` reads databases that hold the displacements of the various parts on a run-by-run basis. The appropriate detector elements are then relocated, and their transforms are updated. As we shall see below, the placement of stubs in global coordinates makes use of these transforms, so these run-by-run geometry updates are important for muon reconstruction.

3.3 Coordinate Systems

In the muon-reconstruction code, several different coordinate systems are referenced. The most familiar of these is of course the CDF coordinate system, which is referred to as the "global" system. Stub positions and directions are stored in their global coordinates, and the `MuonSignedHit` can produce global coordinates on demand.

Two other coordinate systems are useful. One is the "local" coordinate system, which is defined for each chamber. The origin of the system is placed at the middle of the chamber volume, *i.e.* it is on the wire, at the midpoint of the length of the wire. The positive y axis points along the typical muon direction of travel in each chamber, and the z axis is along the wire. The x axis is by construction parallel to the nominal drift direction. Thus, the `getLocalPosition` method of the `MuonHit` as defined above does return a legitimate local coordinate, although by definition it always has a positive x coordinate.

The CMU and CMP both have essentially planar geometries within a given wedge or region. Thus, it is useful for to define a "regional" geometry for those systems for use in pattern recognition. In the CMU, the regional coordinates have their origin at the the geometric center of the wedge volume, with the positive y axis pointing along the typical muon direction of travel, the positive x axis pointing in the direction of increasing stack number, and the z axis pointing along the wires. In the CMP, the axes are oriented in the same manner, with the origin being in the center of the volume that defines a given CMP region[1]. A transformation from local to regional coordinates in the CMU and CMP only requires a translation of the x and y positions of the hit, and never any rotations or change of the z position.

The `MuonDetectorNode`, which holds all of the information about the muon geometry, caches local-to-regional and regional-to-global transforms, and, if they are defined for a given detector element, can be located by `MuonDigiCode`:

```
HepTransform3D locateLocalToRegionalTransform(MuonDigiCode& code);
HepTransform3D locateRegionalToGlobalTransform(MuonDigiCode& code);
```

These transforms are calculated at every begin-run time, as their values might change if the moving parts of the muon geometry are in a different position. Local-to-global transforms for all subsystems are naturally cached by the detector node; it keeps track of the "tangibles" for every detector element by `MuonDigiCode`, and the tangibles carry the transforms. The `MuonSignedHit` calculates its global position by locating the tangible associated with its `MuonDigiCode`, and then applies it

[1] The orientation of the positive direction of the z and x axes are ultimately immaterial; this orientation determines the definition of the drift sign for a hit.

to its local coordinate. Most users will never have to deal with these transforms; they are handled by the various muon data structures.

3.4 Drift Models

The TDC's in the detector measure drift times in the muon chambers, and the D to E algorithms convert this information to drift distances. While this tells us how far the gas ions liberated by the muon passing through the chamber traveled, it does not tell us the path of travel, and thus does not tell us where the muon actually was inside the chamber. To relate the drift distance to a particle position, we need a drift model.

The drift model is typically assumed to depend on the actual location of the particle (which can be taken to be the x position of the particle at $y = 0$ in the chamber's local coordinates), and the "aspect angle," *i.e.* the slope of its path of the chamber. From this information, and a description of the drift properties of the chamber, the expected drift distance can be calculated. Then, a fit to obtain the stub parameters proceeds by taking $x(y = 0)$ (the "geometric" drift distance) for each hit as calculated from the parameters, using a drift model to predict the expected drift distances, and comparing that expectation to the drift distances measured by the TDC's. The parameters are then adjusted to obtain the best match to the measured drift distances. This then changes the predicted drift distances, so an iterative process is required. A good explanation of how drift models work can be found in CDF 4816 [4].

In the muon-reconstruction code, the drift-distance prediction is handled by classes inheriting from `MuonDriftModel`. They have methods that calculate the drift distance given an input geometric drift distance and slope, and also the derivatives of the drift distance with respect to these inputs, which are needed for the stub fit. At the moment, three different drift-model child classes exist. The `TrivialDriftModel` assumes that the ions drift along the local x direction, regardless of the aspect angle. Thus, the calculated drift distance is equal to the geometric drift distance. The `RadialDriftModel` implements the "standard" drift model, where ions drift along the local x direction unless they are within some radius a of the wire, at which point their drift becomes radial. The calculation of the drift distance and derivatives is of course more complicated here than in the trivial model [4]. The `MuonDriftModelA` was derived from the model used for the CMX reconstruction in Run 1, and takes into account changes in the drift at the edges of the chamber.

The line fit for the stubs can depend quite strongly on the drift model used, so the fit residuals will only be sensible if the same drift model is used in their calculation. See the section on the stub-finding algorithms for more details.

4 Databases

The online and offline Oracle databases are used to store those values of calibration constants and detector-position constants which can, in principle, be different for each run. To avoid the need for the user to interface to the database directly there are several classes which read the appropriate constants from the database. The classes have accessors which return the values of the constants. Table 1 lists the names of these classes and the database tables they interface to.

These classes can be separated into two types: those that deal with calibration constants for the muon chambers and muon scintillators (named `XXX_Calibration`) and those that deal with detector-position constants (named `XXX_PositionConstants`). As the constants in the tables are essential for reconstructing muons, if the appropriate tables cannot be found for the current run an ABORT error is issued which will cause your job to crash!

Class Name	Table Names			
CMU_Calibration	CMUAll2	CMUChannel2	CMUWedge	
CMP_Calibration	CMPAll	CMPChannel	CMPStack	CMPRegion
CMX_Calibration	CMXAll	CMXChannel	CMXWedge	CMXQuadrant
BMU_Calibration	BMUAll	BMUChannel		
CSX_Calibration	CSXGlobal	CSXChAST		
BSU_Calibration	BMUAll	BMUChannel		
TSU_Calibration	TSUAll			
CMU_PositionConstants	CMUPosition			
CMP_PositionConstants	CMPPosition			
CMX_PositionConstants	CMXPosition			

Table 1: The classes which interface to the database and the names of the database tables they use.

In case you are curious: it is the responsibility of the `CalibrationManager` to work out which instance of each table is for to be used for each run. Generally there is no need to talk-to the `CalibrationManager`; its default choice will almost always be exactly want you want.

This section describes these how to use the classes in Table 1 and explain which constants they provide.

4.1 Muon-Chamber Calibration Constants

There are four classes which read the muon-chamber calibration constants from the database: `CMU_Calibration`, `CMP_Calibration`, `CMX_Calibration` and `BMU_Calibration`. All of these classes are located in the `Muon` package.

The muon-chamber constants are the drift-model parameter values which are used to calculate a corrected drift time, drift distance, and, if appropriate, a z-position for the hit.

There are six calibration constants for each channel of the muon chamber systems: `timeSlope`, `timeZero`, `driftVelocity`, `offset`, `driftRadius` and `quadCorrection`. In addition the CMU reconstruction requires a total of eight extra constants for each of the CMU channels. Six of these are used z-position of the hit in the chamber: `gainRatio`, `wireRes`, `threshold`, `scaleFactorA`, `scaleFactorB`, `preampRes`. The `timeSlewFactor` and `timeSlewOffset` constants are used to correct the drift time.

There is an accessor for each of the constants which requires the `MuonDigiCode` of the channel as an argument:

- `float timeSlope (const MuonDigiCode& code)`
- `float driftVelocity (const MuonDigiCode& code)`
- `float timeZero (const MuonDigiCode& code)`
- `float offset (const MuonDigiCode& code)`
- `float driftRadius (const MuonDigiCode& code)`
- `float quadCorrection (const MuonDigiCode& code)`

Each of these accessors simply returns the value of that constant for the detector channel specified by `code`. There are similar accessors for `CMU_Calibration` objects for the additional CMU constants.

There is also one other useful accessor for these objects:

- `void allConstants(const MuonDigiCode& code, ...)`

which returns all of the constants for the given `MuonDigiCode`. See the appropriate class header to see in what order the constants are returned.

If an accessor cannot find the channel specified by `code` a default value for that constant will be returned and a warning will be printed.

Please note that the correct `XXX_Calibration` object must be used for the constant you are trying to obtain. For example if you want to use the calibration constants for a CMX channel you must be using a `CMX_Calibration` object. As an example, if you tried to use the `timeSlope` accessor of a `CMU_Calibration` object but the `MuonDigiCode` argument corresponded to a channel in the CMX detector, it would print a warning and return the default value of `timeSlope` for the CMU detector.

4.2 Muon-Scintillator Calibration Constants

There are four classes to which read the calibration constants for the scinitillators from the database: `CSP_Calibration`[2], `CSX_Calibration`, `BSU_Calibration` and `TSU_Calibration`. These classes reside in the `Muon` package.

For each of the muon scintillator systems there are two calibration constants for each detector channel: `timeSlope` and `timeZero`. These constants are used in the appropriate `XXX_DtoEModule` to calculate a corrected time for the scintillator hit.

The accessors to constants also use the `MuonDigiCode` to identify the channel. The accessors to the objects are:

- `float timeSlope (const MuonDigiCode& code)`
- `float timeZero (const MuonDigiCode& code)`
- ```
  void allConstants(const MuonDigiCode& code,
                    float& timeZero, float& timeSlope)
  ```

Again if the channel requested cannot be found a warning is printed and a default value for the constant is returned.

4.3 Muon Position Constants

During the early running periods, large sections of the muon systems were being moved during accesses, and not necessarily returned to their nominal positions. The different positions are also kept in the database, and used in building the geometry description. There are three classes which read muon detector position constants from the database: `CMU_PositionConstants`, `CMP_PositionConstants` and `CMX_PositionConstants`. They are located in the `MuonGeometry` package.

The constants accessed from the `XXX_PositionConstants` objects describe the difference between the real position of the detector and the "default" position described in the offline software. These displacements are used by the `MuonDetectorNode` object to initialize `GEANT` and the

[2] Actually the `CSP_Calibration` class does not yet exist! But when it does it will behave in the way described here.
  ```

CdfTangibles with the detector volumes in the correct positions. In this context the detector volumes refer not just to the CMU, CMP or CMX chambers but also to any attached scintillators and/or shielding.

The accessors to XXX_PositionConstants objects are:

- float xDisplacement(unsigned int key),
  float yDisplacement(unsigned int key),
  float zDisplacement(unsigned int key) – return the displacement in the global x, y or z-direction of the detector volume labeled by key
- Hep3Vector displacementVector(const unsigned int key) – returns the displacement vector in global coordinates of the dectector part labeled by key

The key argument labels the different movable volumes of the detectors. There are have enum values defined for ease of use.

There are four separate pieces of the CMU detector. The defined enum values for key are NE_ARCH=0, SE_ARCH=1, SW_ARCH=2 and NW_ARCH=3 for CMU_PositionConstants objects.

There are also four separate volumes of the CMP detector. For CMP_PositionConstants objects the valid values of key are NORTH_SIDE=0, TOP_SIDE=1, SOUTH_SIDE=2 and BOTTOM_SIDE=3. These do not correspond exactly to the usual definition of the CMP north, south, top and bottom regions; instead, they reflect the way these pieces of the CMP detector are attached together.

Finally there are nine movable volumes of the CMX. The key arguments can be NE_ARCH=0, SE_ARCH=1, SW_ARCH=2, NW_ARCH=3, NE_MINSKIRT=4, SE_MINSKIRT=5, SW_MINSKIRT=6, NW_MINSKIRT=7 and KEYSTONE=8 for CMX_PositionConstants objects.

4.4 Printing Database Values in AC++

It is possible to print out the calibration constants for the muon chambers or muon scintillators in AC++ by talking to the appropriate XXX_DtoEModule. For example to print the CMU constants:

```
talk CMU_DtoEModule
  dumpCalibration set true
exit
```

This will print out the value of all the constants for *all* the channels in that system each time a new run is encountered. This can be a lot of output!

5 Algorithms

All of the AC++ modules that perform the muon reconstruction are really just wrappers for the classes that handle the algorithmic work. The modules handle tasks such as pulling objects out of the event record, and writing new objects back into it, and processing talk-to options. This information is then passed to the objects that perform the real work of D to E conversion, stub finding, and track-stub linking.

5.1 D to E

The classes that handle the D to E conversion in all eight muon subsystems have the name XXX_DtoE, where XXX is the acronym of the subsystem in question. All of them take several different objects as input. A calibration object, described above, provides an interface to the calibration database.

Obviously, the D bank is a crucial input. For simulated events, a set of `OBSP` numbers associated with the hits, which is kept in the event record as an `HitOBSPNumberColl`, is also passed.

For most systems, the D to E process is extremely straightforward. The algorithm is to loop over all the TDC hits in the D bank, extract their start times and widths, and then apply calibration constants. For the chamber systems, these constants include t_0 and linear and quadratic slewing corrections to the start times. A drift velocity is used to convert the time to a distance. For scintillator systems, there is no conversion to a distance; the output is kept as a time. The results are then packed into the appropriate `MuonHit` objects, along with `OBSP` information for simulated events. The `MuonHits` are placed in a flat vector at the start, and once D to E is complete they are sorted by ascending `MuonDigiCode` and either drift distance or time within each `MuonDigiCode`, and packed into the appropriate `MuonHitColl`. The collection is returned to the `AC++` module, which writes it into the event record.

For some systems, more sophisticated processing is required. In the CMU, the hit widths contain meaningful information. Neighboring chambers are ganged together, and the sense wire is resistive, so a muon passing through one chamber will produce hits in the TDC's for both that chamber and its neighbor. The amount of charge that reaches the TDC is encoded into the width of the TDC hit. The resulting charge division allows us to determine the position of the muon along the length of the sense wire [5]; the "real" hit belongs in the chamber which produced the TDC hit with the greater charge. However, retrieving this information requires additional calibration constants, and additional care in handling the input TDC hits.

In a given event, a given pair of ganged CMU chambers might produce N hits on one channel and M hits on the other, leading to $N \times M$ possible pairings. We cannot tell *a priori* which pair is a "real" pair, *i.e.* which hits are caused by real muons and which are the correct companion hits. All combinations must be considered. The formation of each pair is handled by the `CMU_PairSorter` class.

For each pair, calibration constants are applied to the start time and width of each of the hits. The hit with the greater of the two widths (after corrections) is declared to be the "real" hit, and a `MuonXZHit` for the pair will be created in that chamber. The z position of that hit is calculated from the ratio of the two hit widths, and calibration constants such as the resistance of the wire. This z position is measured from the far-from-readout end of the wire; it is then converted to a z position in local coordinates, which have their origin at the center of the chamber. If the hit widths are sufficiently similar in size, it is unclear which of the two chambers the "real" hit belongs in, so `MuonXZHits` are created for each chamber. There is also an option to make `MuonXZHits` out of unpaired TDC hits, but this is turned off by default.

The BMU also has ganged chambers in two of its four layers, and z information is determined. There is an effort to eliminate hits that are believed to come from cross-talk among chambers in the same stack. If there are three or four hits in a stack with similar start times and widths, these hits are marked as being noise and not placed in the final hit collection.

It is also possible to mark the noise bit for hits during D-to-E processing. At the moment, this is only done in the CMP, where any hit that is not the earliest in time on a given wire are marked as noise, to suppress chamber ringing. (This can be turned off with a talk-to command.)

Various talk-to parameters can be set in the `AC++` modules which will lead the D to E algorithms to ignore certain hits. Cuts can be set on the minimum pulse width, minimum pulse start time, and maximum drift distance that are allowed. For the CMU, a cut can be set on the maximum allowed start-time difference between hits in a pair. For the CMX, cuts can be set so that the hit collection is not created if there is a large number of hits, to protect against potential oscillating chambers.

The resulting hit collections from all D to E processes are placed in the event record. Of course, multiple subsystems can write hit collections with the same class name, *e.g.* both the CMP and CMX write a `MuonXHitColl`. The description string is used to distinguish the collections in the event record; each collection is given the description `XXXHits`, where `XXX` is the name of the subsystem.

5.2 Stub Finding

5.2.1 CMU and CMP

The CMU and CMP have many similar features in their geometry. Both can be broken down into regions in which their geometry is strictly planar – the wedge for the CMU, and the region (north, top, south or bottom) for the CMP. Both this fact and the uniformity of the interface for all `MuonHit` objects allows us to use the same algorithm, and in fact the same code, for stub finding in both the CMU and CMP. The only differences are in the values of cuts and parameters, and in the final fit for the stub position and direction.

In this spirit, both CMU and CMP stub finding are handled by the `MuonStubModule`, which passes data to a `MuonStubFinder` object. The module constructs a `MuonDriftModel` for each subsystem. For both CMU and CMP, the `RadialDriftModel` is used, and the radius a is extracted from the calibration database. The current default radii are 1.6 cm for the CMU and 0.8 cm for the CMP. These drift models are then used to initialize stub fitters for each subsystem; the fitters are discussed below. The module also extracts the hit collections with the appropriate descriptions from the event record.

The `MuonStubFinder` works off the hit collection and the fitter, and returns a `MuonStubColl` that can be placed in the event record at the end of processing. The stub finder loops over the appropriately-sized regions (wedges for the CMU, regions for the CMP). In each region, the finder counts the number of hits in that region, and the number of layers that have hits on them. Hits that have been marked as noise hits are ignored here, and throughout the stub-finding process. If there are fewer than three hits in the region, or hits exist on fewer than three layers, then there is no reason to proceed, and the finder moves on to the next region. If there are a large number of hits in the region (20 for the CMU, 250 for the CMP, modifiable by talk-to), the finder logs an error message and does not try to find stubs, out of fear that too much CPU time will be consumed.

Once the finder has chosen a region to search for stubs, it forms four vectors of `MuonSignedHits`, one for each detector layer in that region, containing all of the signed hits in that region and layer. The finder then examines all pairs of hits taken from layers 0 and 2 that are within a specified distance of each other in regional x (7.5 cm for CMU, 12 cm for CMP, modifiable by talk-to). These two hits are used as the basis for a line segment; the slope and intercept are calculated in regional coordinates. No drift model is used at this point, so the line parameters are only rough estimates. The line parameters are used to estimate the location of hits in layers 1 and 3. The finder then looks for hits in those layers that are within a specified distance (0.5 cm for CMU, 1.5 cm for CMP, modifiable by talk-to) from the prediction in regional x. If both layers 1 and 3 have hits falling in this road, then four-hit stubs will be produced, and the finder will not even try to make three-hit stubs. All combinations of hits from these two layers will be used to make candidate stubs. If only one of those layers has hits in the road, then three-hit stubs will be made. In this case, only one hit needs to be examined – the one that is closest to the prediction, as all others will surely give a worse fit. This entire procedure is then repeated, using layers 1 and 3 as the starting layers, and then searching for hits in layers 0 and 2. If four-hit stubs were found in the first pass, then there is no need to create them again in the second pass. These collections of hits are stored in a `CentralMuonStub` object for further processing.

This gives us a potentially large number of candidate stubs to examine. The finder then uses the fitter that was passed to it to fit the stubs, using the drift model that had been supplied to the fitter. In the CMU, two independent fits for the stub parameters are performed, one in the $x - y$ plane and one in the $x - z$ plane, where we refer to the regional coordinates in both cases. The $x - z$ fit is a standard linear least-squares fit. It does not include hits that are marked as having an invalid z position or have a TDC width of 2047; this overflow value in the width makes the charge-division calculation unreliable, and these hits do not have useful z information. It is possible for a stub to have no hits with reliable z information; in that case, the stub will be placed in the middle of the chamber in z. The χ^2 values for the $x - z$ and $x - y$ fits are stored separately in the `MuonStub`. In the CMP, there is no z information, so the stub is placed at $z = 0$ in local coordinates.

The $x - y$ fit is more complicated, because of the drift model. This is because the locations of the hits change with the stub aspect angle. When line parameters are determined, they specify an aspect angle which may be different from the one which was used to determine them. Thus, an iterative procedure is required. The fitter starts with a simple least-squares fit, to get the first estimate of the intercept and slope of the line. The line segment is then translated back to local coordinates for each of the hits used into the fit, which allows for a measure of the "geometric" drift distance in that chamber. From that, the line slope, and the drift model, the drift distance is predicted. The prediction is compared to the measured value determined at D to E time, and a χ^2 is formed and minimized. This procedure is repeated until the change in χ^2 between iterations is small, or ten iterations have been performed. The final parameters, along with their errors and the final χ^2, are written into the stub in regional coordinates.

Both the CMU and CMP fitters inherit from `CentralMuonStubFit`, which specifies the interface. The CMU uses the `CentralMuon2DStubFit`, and the CMP uses the `CentralMuon1DStubFit`, which differ only in that the former does the $x - z$ fit (which is of no use in the CMP), and each fitter calculates derivatives according to the coordinate system of the appropriate chamber. The least-squares fit is done using classes from the `Fitting` package, and `MuonFittingHit` and `MuonFittingStub` objects with the appropriate interfaces for those classes are created.

The true residuals of this fit, then, are the differences not between the hit positions in local coordinates and the stub position, but between the drift distances of the hits and the predicted drift distances based on the stub parameters. To calculate a proper residual between a hit and a stub, the `getStubResidual` method of `MuonSignedHit` should be used. It takes a `MuonStub` as an argument, and the signed hit is *not* required to be part of that stub; thus, this method can be used to find the residuals between arbitrary pairs of hits and stubs. A `MuonDriftModel` can be passed to this method as an optional argument. If it is, the residual returned will be the difference in the measured and predicted drift distances. If it is not, the trivial drift model is assumed, and the value returned is simply the difference between the hit and stub positions in local coordinates.

These fits are performed on every candidate `CentralMuonStub` that was found five paragraphs ago. The stubs are then sorted by the number of hits and the $x-y$ fit χ^2. Four-hit stubs are ranked ahead of three-hit stubs, and within those classes stubs with smaller χ^2 are ranked ahead of those with larger χ^2. (In the CMU, the χ^2 contribution from the $x - z$ fit is *not* currently used in this ranking.) Then the finder tries to determine which stubs on this list are duplicates of other stubs. If two stubs have at least two `MuonHits` in common, then the stubs are considered duplicates, and the lower-ranking stub is eliminated. Note that that the comparison is on the `MuonHits` and not the `MuonSignedHits`; the drift signs are ignored for this purpose, and only the underlying TDC hit is examined.

The surviving stubs then converted into `MuonStub` objects, and their fit parameters are converted from regional to global coordinates, using the transforms cached in the `MuonDetectorNode`. These are then placed into MuonStubColls – one for the CMU, which is given the description CMUStubs, and one for the CMP, which is given the description CMPStubs. The module writes the collections into the event record.

5.2.2 CMX

The CMX_EtoSModule reconstructs stubs using hits from the CMX drift chambers. There are two major steps. First, the pattern recognition algorithm locates stubs and determines which hits are associated with the stub. Second, the fitter determines the precise location and direction of each stub.

Input and Output To get its input, the module looks for objects of class MuonXHitColl with description CMXHits. As described above, by default it will only look for hit collections created in the current process. If the talk-to option findProcessName is set true, it will look for the process name specified in the talk-to option processName. This defaults to PROD (the production process). The module also uses CSX hits for some limited purposes. It looks for CSX hits in a MuonTHitColl object with description CSXHits from the same process as the CMX hits.

There is another option for input that most people will never need to use. If the talk-to options readCMXEandCreateHitColl and/or readCSXEandCreateHitColl are set true, then the module will look for the newest CMXE bank and/or CSXE bank in the event and create hit collections from them. The hit collections are stored in the event and then used to find stubs normally. Some of the information in these hit collections is missing (widths and TDC words), but these quantities are not currently used in the stub reconstruction. One can use these options to process Run 1 CMX data or very old simulation data with CMXE/CSXE banks. When processing Run 1 data, one should also set the run1 talk-to parameter true.

The other input to the algorithmic code comes from the geometry. The fitter uses the transforms from the MuonDetectorNode to know the locations of the chambers and precisely calculate the position and direction of each stub. These are the transforms from the CdfTangibles that are corrected for alignment on a run by run basis using values from the database. The fitter can handle chambers that have been individually aligned. The pattern recognition and other parts of the module use geometric constants that cannot be corrected for alignment. These constants reflect the nominal local geometry and come from the class defined in MuonGeometry/CMXConstants.hh. These same constants are used to build MuonDetectorNode and are consistent with the transforms in the CdfTangibles before alignment corrections. Alignment problems at the granularity of the 9 major parts of the CMX (4 arches, 4 quadrants of the miniskirt, keystone) should have no effect on the pattern recognition, but alignments at a smaller granularity will cause problems if they are large enough.

As output, the module produces a MuonStubColl that contains MuonStub objects for the CMX. The collection has the description CMXStubs. By default, this collection is written into the event record, but this can be disabled by setting the talk-to option writeMuonStubColl to false. There is another output option that is greatly deprecated and may some day be deleted. If one sets the option writeStubBanks true, then a CMUS and CMUR bank will be written into the event record for each stub. By default this option is set false. This is the old Run 1 format. Its only current advantages are that it holds residuals directly from the fit and CSX information which the MuonStub objects lack.

There are two things that can cause hits to be ignored immediately after being read from the bank. Such hits are not used during pattern recognition. If the talk-to option ignoreNoise is set true, hits flagged as noise in the hit collection are ignored. By default this is false. The talk-to option `maxHitsPerChannel` contains a limit on the number of hits from a single chamber. If the limit is exceeded, *all* hits from that chamber are ignored. By default, this is set at 7. Chambers that are oscillating and producing 8 hits in every event are ignored. Without this cut, oscillating chambers cause the stub finding to take a long time and produce many junk stubs.

There are a few checks made on the input hits. The `MuonDigiCode`s must identify legal chambers that exist. Within a channel, the hits need to be time ordered. All hits for a single channel must be sequential. If the input hit collection was created by the standard D to E module, then all these things will automatically always be true. The D to E module takes care of sorting and reports illegal channel addresses through the error logger (and deletes the hits).

CMX Pattern Recognition The pattern recognition algorithm is based on the Run 1 algorithm. There were many modifications and improvements to the algorithm, but its basic form and many of the details are the same. In addition to the algorithmic changes, the code was completely rewritten in C++. CDF note 2871 [6] documents the Run 1 algorithm and has some interesting studies on Run 1 data. We expect to achieve an efficiency and global ϕ measurements as good as in Run 1. There should be improvements in the measurement of the z coordinate. The algorithm was also modified to accomodate the complicated miniskirt geometry which is quite different than the arch geometry. CDF 5079 [7] describes the miniskirt drift-chamber geometry and CDF 5055 [8] describes arch drift-chamber geometry.

The new algorithm has been tested on Run 1 data and Run 2 simulation. Parameter values are the same as in Run 1 or were tested using Run 1 data. No careful tests have been done with miniskirt data yet.

The algorithm works as follows:

1. Loop over seeds. A seed is pair of two chambers located at the same ϕ. Require both chambers in the seed to have at least one hit. Skip seeds marked as used. (Note the trigger also requires a pair of hits from two chambers at the same ϕ.)

2. Get the drift distances of hits for 14 chambers: the two seed chambers, the 6 nearest chambers at higher ϕ, the 6 nearest chambers at lower ϕ. Correct these with a drift model that assumes radial tracks.

3. Loop over all possible pairs of hits with one hit from each of the two seed chambers. Also loop over the 4 possible drift sign assignments for the two hits.

4. Require the change in drift distance per layer to be less than 0.58 cm. There is a talk-to parameter named `maxAlpha` that allows one to modify this cutoff. Note that the layer separation is about 2.7 cm.

5. Get the hits that lie in the 12 nearest chambers that could fall on a road centered on the two seed hits. The road width is 4 mm plus an amount that allows for the possible variation in wire separation for these chambers. The 4 mm can be modified using the talk-to variable `hitTolerance`. The allowed range of wire separation varies for different chambers.

6. The 4 drift sign possibilities are divided into 2 categories: either the two drift distances have the same sign or the two drift distances have opposite sign. Within those categories choose the drift sign for the inner hit by selecting the sign associated with the largest number of chambers with at least one hit on the road.

7. Look for a second seed where both chambers have at least one hit on the road determined by the first seed. Loop over all such possibilities. Require the change in drift distance per layer to be less than 0.58 (`maxAlpha`) and also the difference between this and the slope for the first pair to be less than 0.15, a talk-to parameter named `maxDeltaAlpha`.

8. With two pairs of hits, one can calculate the separation between the wires. Do it and then repeat the step of looking for addition hits along the road, only now the road width is simply 4 mm (`hitTolerance`) because the wire separation is known.

9. In cases where a second seed (chamber pair) cannot be found, use a single additional hit. With a single hit and a seed the wire separation can also be calculated. Loop over all such hits and for each try to add additional hits along a road with a width of 4 mm (`hitTolerance`).

10. For each seed, either zero or one stub will be returned by the pattern recognition function. If more than one candidate passing the cuts is found, the best is selected from all the possibilities in the loops described above using the following criteria.

    - Stubs with two pairs of hits are always selected over other stubs.
    - More hits is better.
    - For stubs with the same number of hits, a quality figure based on the distance of hits from the road is calculated. Hits used to determine the road are weighted differently than the other hits to better approximate a χ^2-like value. A small penalty is added to this quality figure for stubs missing the CMX in the z direction. This penalty increases as the distance between the stub and the CMX increases. The best-quality stub is selected.

There are a number of geometric constants and correction factors that account for the geometry of the arches and miniskirt that are not described in detail above. These allow the same algorithm and code to be used for both parts of the CMX.

There is an explicit cut in the pattern recognition that prevents hits from an arch and the keystone on the same stub. This creates a gap roughly 10 to 15 cm wide in the acceptance between the SW arch and the keystone. This cut has been made based on the assumption that the alignment of the keystone and arch will not be good enough or consistent enough to make stubs across the boundary. The arch rests on the floor and the keystone hangs from the ceiling. They are not connected. Near this boundary there are 3 chambers that exist that will never be used in a stub and some others that will not be used as often as one would expect. This is not an issue on the NW because there are 4 chambers that are physically missing at the NW arch-keystone boundary and the acceptance gap is unavoidable.

The pattern recognition function will return a stub with between 3 and 8 hits (actually if the `hitTolerance` is increased to a large value, 9 or 10 hits is also algorithmically possible even though not physically possible).

After the CMX Pattern Recognition The list of hits associated with a stub is passed to a fitter function. The fitter uses the hits and the alignment-corrected transformations (the chamber positions) from the geometry to determine the stub position and direction. It minimizes a χ^2-like quantity that is based on the residuals. The fitter calculates the position of the stub in CDF global coordinates. It also calculates the direction of the stub in the plane that includes vector pointing to the origin and a vector pointing in the direction of increasing ϕ. It does not determine the other direction from the fit; dr/dz is assumed equal to r/z (where r is cylindrical radius). This assumption is as good as trying to determine this direction from the fit. With this assumption the fit will converge with only 3 hits on the stub, otherwise a minimum of 4 hits would be needed.

The `MuonDriftModelA` (same drift model as in Run 1) is used to calculate the residuals used in the fit. In this model, the electrons within `driftModelRadius` (a talk-to parameter that defaults to 0.8 cm) of the sense wire, drift radially straight to the wire. Outside that radius they drift in the local x direction. There is an additional nonlinear correction applied that accounts for the fact that the drift velocities are not constant in the chamber.

While it is possible for stubs to fail the fit and be lost, this almost never happens unless the parameters in the pattern recognition are modified. There are some cuts that restrict the z coordinate in the fit and if the stub favors a position very far from the CMX it may fail the fit. Increasing `hitTolerance` used in the pattern recognition can cause this to happen often.

The function that does the fit has been rewritten in C++, but is mathematically identical to the fitter used in Run 1. One shortcoming is that it does not return errors on the fit parameters, but these would not be very useful anyway. CDF 4815 [9] documents many of the details of the fitter and the math used there. Beware if you read that note. There are about a dozen errors in the formulas and text in the note. Even with the errors, it explains the concepts well and a careful reader could spot and fix the errors.

There is a talk-to parameter called `sigma` which has a default value of 0.05 cm. This parameter is used as an error on the drift distances when calculating the χ^2-like quantity that is used in the fit and returned by the fitter. This value is at best a very rough estimate of the errors on the hits. The χ^2 is not normalized very well.

There are four requirements made on the stubs after the fit is completed. The χ^2-like quantity must be less than the talk-to parameter `maxCHI`. The default value is so high this cut normally does nothing. The absolute value of $r(d\phi/dz)$ (called `XI` in the code) must be less than `maxXI`. The default value of this cut is 0.5. `XI` is highly correlated with `Alpha` that is cut on in the pattern recognition. The default value of `maxXI` does little because the cut `maxAlpha` is somewhat tighter (at least by default). `minAR` and `maxAR` cut on dr/dz of the track. It must be greater than `minAR` and less than `maxAR`. Since dr/dz equals r/z, this is equivalent to a fiducial cut. The default values are r/z values well outside the limits of the CMX. If the fit prefers a position far from the CMX this cut will remove them. There are cuts in the pattern recognition based on wire separation and `hitTolerance` that are strongly correlated with this cut.

By default, the CSX scintillator hits have no effect on the output of the CMX stub finding. There are talk-to parameters that allow one to require stubs have matching CSX hits. Setting `requireMeantimeHitsCSX` true requires a CSX meantime hit be matched to a stub before it is written to the event. `nLayersRequiredRawCSX` can be set to 1 or 2 and requires 1 or 2 raw CSX hits (not meantime hits). `nCMXHitsForNoCSXHitsCuts` allows one to only require CSX hits for stubs that have a small number of CMX hits. There are two parameters that affect the matching of CSX hits to stubs. `maxTimeCSX` will cause CSX hits with greater than the time to be ignored. `matchToleranceCSX` is by default 4 cm. This is the maximum distance from idealized scintillator counter boundaries that are not quite correct. 4 cm is large enough to account for small alignment problems and places on the miniskirt where the actual scintillator boundaries are not exactly where the idealized model places them. There are some runs where the arches were not in position. In these runs, the function that matches CSX hits will not match things properly unless the `matchToleranceCSX` is increased. This part of the code is not presently smart enough to look this up in the database. One last comment about the CSX code: the function that performs the matching is a static standalone function and can be run independent of the CMX stub finding. See `CMX_EtoS::matchScintillatorHits`.

For stubs that pass all the cuts, all pairs of chambers at the same ϕ where both chambers have a hit associated with the stub are marked as used seeds. Although hits in these used chambers can be reused in other stubs, they will not be used as a starting point (seed) in later pattern recognition.

As stubs are made during the main loop over the seeds, they are temporarily stored in a vector. As each is added, it is checked against the other existing stubs and duplicates are deleted. If two stubs share 3 or more hits, then they are considered duplicates. When there are duplicates, the inferior stub is deleted. The inferior stub has fewer hits. If the number of hits is the same, then lower χ^2 is considered inferior. The surviving stubs are written into a `MuonStubColl` that is later put in the event record.

Immediately before being written to the `MuonStubColl`, there is a short straight-line extrapolation done along the direction of the stub such that the position of the stub always lies on the same surfaces. For arch and keystone stubs, these surfaces are cones, one east cone and one west cone. For the miniskirt stubs, the surfaces are planes, one east and one west. The surfaces lie at the center of the detector layers in their ideal nominal positions. The surfaces do not move with chamber alignment corrections, which simplifies the linking of the stubs to tracks. The tracks are extrapolated to these same constant surfaces.

5.2.3 BMU, BSU, TSU

For the BMU reconstruction, the D to E module has produced `MuonXZHits` which are "real" and account for the ganged cells, and which have z information when available. These hits also use the survey alignments for the wire positions. The z information is not used in the stub finding itself, though this can be used in determining stub quality later. The drift model for stub-finding is currently a simple linear one.

As of this writing, the scintillator information from the BSU and TSU is not yet integrated into the final `MuonStub` object. This improvement will be available shortly.

There are two independent stub finders: one written well before the `AC++` framework was established by James Bellinger, and the other written as a standalone analysis in FORTRAN by Robert Handler (and translated into C++ by James). Each is somewhat tunable, and as of now the sets of stubs they find do not entirely overlap. The latter package attempts to optimize the cuts using t_0 and the drift speed to avoid contamination from halo and splash and other buckets. A talk-to parameter decides which algorithm will be used; the Bellinger algorithm is the current default.

Bellinger Algorithm The Bellinger algorithm begins by grouping the hits by layer for each side. In order to simplify the logic it uses a trick: since there are no chambers in the lower quarter of the toroid, the hits are sorted by ϕ' rather than ϕ, where the $\phi' = 0$ axis points downward. This avoids wraparound tests in the code.

The stub finding operates on collections of hits, one (or zero) per layer, which are managed by `BMU_Scan`. `BMU_Scan` checks for "no more hits," makes sure there are enough hits in the specified ϕ window, and generates the *next* collection by finding the hit with the lowest ϕ' and picking the next hit from that layer (iteratively if necessary). Using a ϕ window ensures that the stubs point more or less back to the beamline, though the window is currently rather wide. At least one point in the collection must have a good z estimate.

Once we have a collection of hits, we estimate a t_0 using an idealized geometry, and correct the positions of the hits in space using the presumed arrival time of the track. Then we translate and rotate to the center of the collection of hits, and fit the points using a simple two-parameter χ^2 fit. There is a second three-parameter fit for the t_0 as well, but this has not been as stable as hoped.

This defines a candidate BMU stub. It is compared to any other existing candidates to see if it overlaps, and if so we determine which is the best on the basis of the χ^2. The maximum limit on χ^2 and the difference in χ^2 required to move from four-hit to three-hit stubs are settable at run time.

When the scan is complete, `BMU_StubModule` receives a collection of `BMU_Candidate` objects, and translates those that pass specified requirements (t_0 cuts and so on) into `MuonStub` objects, which are placed in the event record.

Handler Algorithm This algorithm is based on looking at hits in half-cells. A half-cell is a virtual tower of cells, which may be even or odd depending on whether the sides of the cells examined are all in the same stack or in interleaved stacks. The first thing the algorithm does is generate a collection of all half-cell towers which have hits. A single cell will contribute to two different half-cell towers.

Within a single half-cell tower, we look first at combinations of four hits, then later at three-hit combinations if no four-hit stubs were found. Using the drift times, corrected for the ideal geometry as needed, we extract the t_0 (using a best-guess for three-hit combinations), skip the stub if the t_0 is not in the expected range, and then use the time differences to determine the stub quality. We then check to see if the hit with the other drift sign (on other side of the wire half-cell) is more appropriate, replacing hits as needed.

When done, the correct positions in space are generated, and the same fitting as in the Bellinger algorithm is done to generate `BMU_Candidate` objects. As with the other algorithm, when all is scanned, `BMU_StubModule` receives a collection of `BMU_Candidate` objects, and translates those that pass specified requirements into `MuonStub` objects, and places them in the event record.

Scintillator Matching This is still in development, since the exact relationships of the timing windows depend on the t_0 used in `BMU_DtoE`, which depends on the drift velocity.

The plan is simply to look at the scintillator counters which are pointed at by the stub within tolerances (for the BSU) or in the same phi within tolerances (for the TSU). If a counter has a hit which agrees within the timing window with the t_0 found for the `BMU_Candidate`, then that hit will be added to the list of hits in the `MuonStub`. Having more than one hit within the window is treated as if there were no hits.

As of now we do not plan to demand the presence of a scintillator hit (or more than one since we can have front BSU + rear BSU + z near the center of the BMU, or rear BSU + TSU) for a `MuonStub`, but to use them to express the quality of the stub.

5.3 Track-Stub Linking

The track-stub linking algorithm attempts to use the reconstructed stubs to identify which tracks (reconstructed in the SVX, COT, or both) are most likely to belong to muons. This is done by extrapolating the tracks to the stubs, through the inhomogeneous magnetic field and dense material, and seeing whether they are close to each other. Of course, more than one track may make a reasonable match with a stub, and a muon might travel through more than one muon subsystem, so a given track may need to have more than one stub attached to it. Thus, some amount of arbitration must take place among the possible track-stub pairs, to determine which are the "best."

5.3.1 Input Objects

As with the other algorithmic code, the inputs for the algorithm are selected at the module level, which in this case is the `MuonLinkerModule`, and then passed to an object that does the work, which in this case is the MuonLinker. The obvious inputs are the stubs and tracks that are to be matched. All the MuonStubColls (with the correct process name) are read in from the event record, and the collections are gathered into a vector which gets passed to the MuonLinker. Tracks are selected by creating a CdfTrackView through the defTracks method of that class, using the chosen process name. This view is then filtered, removing tracks with low momentum ($p_T < 1.3$ GeV/c), large impact parameter ($|d_0| > 6$ cm), large z at the distance of closest approach to the beamline ($|z_0| > 200$ cm), and few axial COT hits ($N_{ax} < 10$). These values can be changed by talk-tos in the MuonLinkerModule. The contents of the resulting CdfTrackView is passed to the MuonLinker.

The MuonLinker also requires two extrapolators, and a "criterion" object for use in arbitrating the track-stub pairs, as discussed below.

5.3.2 Extrapolators

The extrapolators must inherit from the AbsExtrapolator class in the Extrapolator package; the two extrapolators that are currently available are described below. Which extrapolators to use as the "first" and "second" extrapolators can be chosen at run time *via* talk-tos; if more extrapolators are developed, they too can be added to the module for selection *via* talk-to.

The extrapolators must be constructed with CdfDetector* as an argument, should they need to reference the detector geometry. They are allowed to carry talk-to options which can then be appended to the MuonLinkerModule. To use an extrapolator, it must be initialized with a particle's identity, using the methods setParticleType (taking the PDG particle code as an argument), or setParticleCharge and setParticleMass. Then, the initial position and momentum must be initialized, *via* the methods setInitialPosition(HepPoint3D& xstart) and setInitialMomentum(HepVector3D& pstart).

The extrapolator then knows everything it needs to about the particle, whose path can then be predicted. It can be extrapolated different specified geometries, *via* extrapolateToRadius, extrapolateToZ (taking a single floating-point number as an argument), extrapolateToCone (taking the cone apex and slope as arguments), and extrapolateToPlane (taking a HepPlane3D as an argument). The extrapolator returns error codes if there is any difficulty reaching that geometry.

Once extrapolation is complete, then the current status of the particle can be obtained *via* the getCurrentPosition and getCurrentMomentum methods. Other methods will return the amount of material that has been traversed, the energy lost by the particle, and the uncertainties on the position and momentum.

Most users will not need to operate any extrapolator directly. Instead, the MuonExtrapTool class can be put to use for most purposes. The tool can take either a CdfTrack or a CdfMuon as an argument, and a pointer to a specified extrapolator as an optional argument. It will return a boolean value indicating whether the extrapolation was successful, and the specified extrapolator can be queried for information on the current state of the particle, as described in the previous paragraph. If no extrapolator is specified, then the tool will use one of its own choosing; this will work, but the particle's current-state information will be unavailable, and the boolean is the only useful return value. The tool knows how to extrapolate to a particular MuonStub, or to various locations in the calorimeter. The MuonExtrapTool is used extensively in the creation of a CdfMuon, and more details will be provided below. In addition, a user can use it to do extrapolations for their own purposes.

At the moment, there are two extrapolators available for use – the MExtrapolator and the G3XExtrapolator. Their functionality is discussed below. The MExtrapolator is currently used as both the "first" and the "second" extrapolator. To use G3XExtrapolator as the "second" extrapolator, see the instructions in Appendix B.

Simple Fast Extrapolator The MExtrapolator uses a simplified geometry model for extrapolation. MExtrapolator does not account for material interactions *i.e.* energy loss and multiple scattering. The main purpose is to provide the solution for a particle's motion in a non-uniform magnetic field with appropriate precision. The general interface is inherited from AbsExtrapolator. The extrapolation methods are:

- extrapolateToRadius, extrapolateToPlane, extrapolateToZ, and extrapolateToCone methods generally assume the starting point for the extrapolation is inside the volume confined by outer COT borders ($|z| < 160$ cm, $r < 149$ cm) and initially moving out of the detector. For other starting points these methods might fail and will return an error. The extrapolateTo methods properly account (with good precision) for magnetic field changes between the initial and final extrapolation points.

- Additional (publicly accessible) utility methods are moveToRadius, moveToZ, moveToCone, moveToPlane, and stepInNonUniform. These allow for one step extrapolation from an initial point (defined as for extrapolateTo* with setInitialPosition and setInitialMomentum, or using the previous extrapolation results; no initialization needs to be done in this case) to the final point defined by radius, z, cone, plane, and step length respectively. These methods assume no magnetic field change along the extrapolation step.

  - The moveTo* methods account only for non-zero z component of the magnetic field and require an InitTrajectory call (with HELIX or LINE used as arguments) for proper extrapolation, HELIX is used if $B_z \neq 0$ for the magnetic field along the step; LINE is used if the field is supposed to be zero.

  - stepInNonUniform requires no InitTrajectory call and performs a one-step extrapolation in an arbitrarily-directed magnetic field (assuming that the field is uniform along the step).

Internally, the extrapolateTo* methods make use of moveTo* and stepInNonUniform to perform the extrapolation. These methods rely on classes from the Trajectory package.

Various methods required by the AbsExtrapolator interface return null or unusual results, by construction. getRadLengthsTraversed returns the path length in cm, rather than radiation lengths. All the methods that are meant to return other information related to material integration, or uncertainties on the extrapolation, consistently return zero, since MExtrapolator does no material integration.

MExtrapolator performs well. Matching quantities for sufficiently large p_T (3 - 5 GeV/c) do not show a significant p_T dependence.

GEANT-based Extrapolator The G3XExtrapolator uses GEANT [11] to step through the geometry description. The use of a single official geometry throughout the offline is a major advantage. This way geometry changes propagate transparently to both reconstruction and simulation. The access to material data in the GEANT framework allows for some additional features.

The GEANT engine and geometry has to be initialized for the extrapolator to work successfully. This is accomplished by the SimInitManager module. Accordingly this module has to be included into the executable. An example G3Xtestm.cc to illustrate usage is included in the Extrapolator package. A script file g3xtest.tcl is also supplied.

The general interface is that inherited from `AbsExtrapolator`. Additional methods include the ability to extrapolate to a given volume, where the volume is identified via its four-character `GEANT` name.

The main motivation for a `GEANT`-based extrapolator is the ability to retrieve information about the traversed material. This can be used to estimate extrapolation errors. The estimated errors provide a very powerful χ^2 muon-matching quality discrimination variable. For this calculation formulas derived in CDF 1430 [10] have been used. The accessors `getRadLengthsTraversed`, `getAbsLengthsTraversed`, `getLengthsTraversed` and `getEnergyLost` provide the raw information to calculate the χ^2 values. The methods `getXErrorMatrix` and `getZErrorMatrix` each return a two-dimensional matrix. The elements can be accessed via the methods `getSigmaXIntercept`, `getSigmaXSlope`, `getSigmaZIntercept` and `getSigmaZSlope`.

The routine `getMaterialList` returns a vector of `stepinfo` structures that contain the location and material for each step. A detailed description can be found in the `G3XExtrapolator` header file.

Internally, the extrapolator makes use of modified GEANT user routines `gustep` and `gukine`. The `gustep` routine is responsible for the handling of each extrapolation step performed by the `GEANT` engine. It extracts at each step material information from the `GEANT` commons and checks if the selected termination condition has been reached. If the particle is close to a terminating boundary (*e.g.* the extrapolation radius), the size of the next step is adapted to not overstep that boundary. Modifying `gukine` allows for particle start-data input independent of the CDF event structure. The previously-set particle type, starting point and and direction are transfered into appropriate `GEANT` commons.

The `G3XExtrapolator` performs well. No deviations in the matching quantities have been found. Figure 1 shows the Δx distribution in the CMU as a function of muon momentum; we see the appropriate narrowing with increasing momentum. The error estimation has been successfully tested for the CMU system on J/ψ data. The pull distribution is shown in Figure 2. The distribution is nicely centered around zero and its width is comperable to unity. The pull distribution is independent of momentum, demonstrating that the extrapolator is estimating the errors correctly. Due to lack of statistics the behavior for the CMX system has been studied on a Monte Carlo sample. Errors are slightly underestimated, but the distribution is satisfactory.

5.3.3 Criterion

The criterion object gives information about the quality of the track-stub match that is used later on to decide which stubs go with which tracks. It must inherit from the `MuonMatchCriterion` class. This class has a method called `pass`, which takes a `CdfTrackExtrapolation` (see below), a `MuonStub`, and a `MuonMatchData` object as input arguments. It returns a boolean which says whether this track-stub pair should be considered at all, and a "quality value" q, a number that is a measure of the quality of the track-stub match. The smaller the value of q, the better the match; it is meant to be a χ^2-like quantity. In principle, many classes inheriting from `MuonMatchCriterion` can be written, and then selected *via* talk-tos for use in the track-stub linking. But at the moment, there is only one such class, the `MuonPhiMatchCriterion`. This class uses the absolute value of `dx` from the `MuonMatchData` as q. It makes additional checks based on which subsystem the stub is in. These are:

- CMU: $q < 30$ cm, $|z_{track}| < 250$ cm, track and stub on same side (east/west) unless $|z_{track}| < 20$ cm
- CMP: $q < 30$ cm, $|z_{track}| < 330$ cm

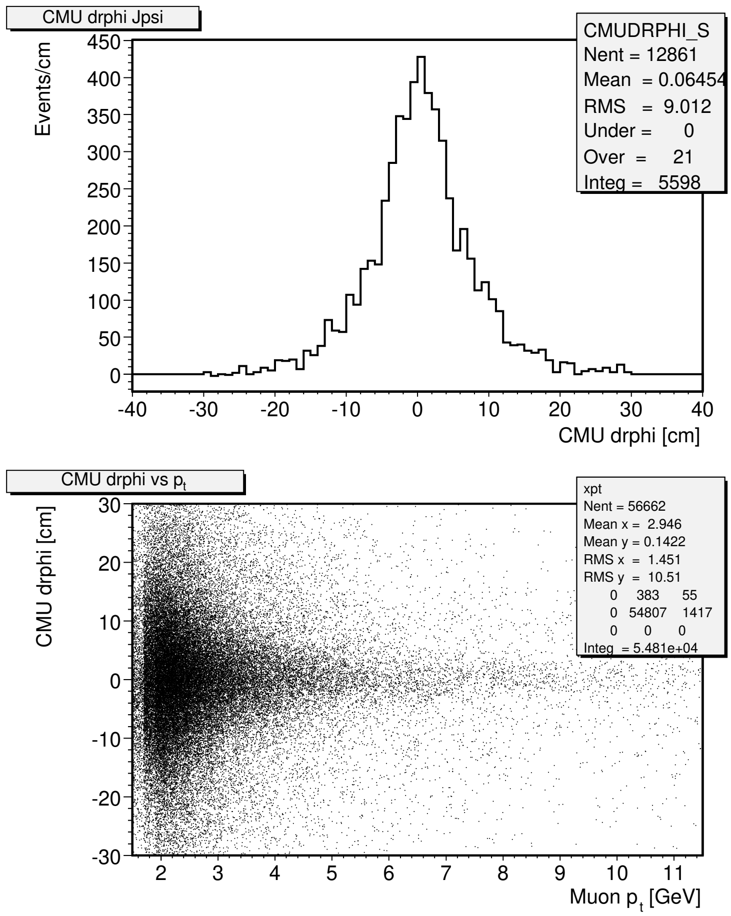

Figure 1: The distribution of the matching quantity Δx for the CMU system.

- CMX: $q < 50$ cm, $325 < |z_{track}| < 550$ cm, track and stub on same side
- BMU: $q < 90$ cm, $440 < |z_{track}| < 840$ cm, track and stub on same side

Here, $|z_{track}|$ always refers to the global z position of the extrapolated track. All of these values can be modified *via* talk-tos in the `MuonLinkerModule`. If a track-stub pair fails these requirements, the criterion returns false, and the pair will not be in a `CdfMuon`.

5.3.4 Track-Stub Arbitration

Once the stubs, tracks, extrapolators, and criterion have been specified to the `MuonLinker`, track-stub association and arbitration can take place. An important rule of the association is that, in the end, a stub is not allowed to be associated with more than one track. So, we start by selecting the tracks that might end up being associated with each particular stub. For each stub, we create a `MuonTrackCandidates` object which holds this information. This object keeps track of which `MuonStubColl` the stub belongs to, to make the correct association in the final `CdfMuon`, and it will frequently be referred to as "the candidate."

Then, each track is extrapolated to each stub, using the `MuonExtrapTool`. For the CMU, CMP and BMU, the track is taken to the radius (in cylindrical coordinates) of the stub; for the CMX, it is taken to the CMX cone for stubs in the arches and keystone, and to the miniskirt plane for

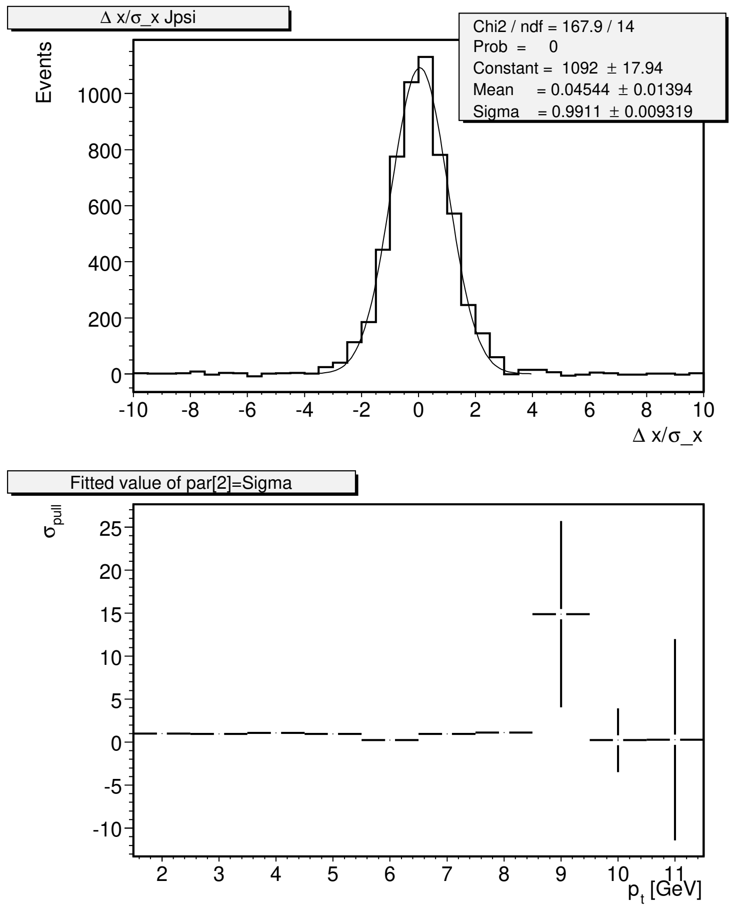

Figure 2: The distribution of the pull $\Delta x_{CMU}/\sigma_{est}$ for sideband-subtracted J/ψ data. In the lower plot the momentum dependence of the width of this distribution is shown. The last bins suffer from low statistics.

stubs in that region. At this stage the "first" extrapolator is used in the tool. If the extrapolation is successful, we create a `CdfTrackExtrapolation` object for the track. It holds a pointer to the original `CdfTrack` and the information about the final state of the particle, such as the position and momentum, as obtained from the extrapolator.

This `CdfTrackExtrapolation` is then added to the `MuonTrackCandidates` object for the stub in question, with the criterion used as part of that operation. That sentence actually covers a number of operations. The `CdfTrackExtrapolation` and the stub (which is living inside the `MuonTrackCandidates` object) are used to calculate the `MuonMatchData` for the track-stub pair. This data structure was discussed above; it holds all the match-quality information of interest. This calculation is done by the `MuonMatchCalculator` class, which is just a way to have all of these calculations appear in one place in the code. Now, with the track extrapolation, stub, match data and criterion in hand, we can use the criterion to decide if this track-stub pair should be considered for use in a `CdfMuon`. If (and only if) the pair passes the criterion, the track extrapolation, match data, and quality value q are appended to a list in the `MuonTrackCandidates` object, for future consideration.

Once this has been done for all tracks, we have a set of `MuonTrackCandidates` objects each of which now holds a list of tracks that may be good matches to a stub. We remove candidate objects that have no tracks; their stubs will not end up in muons. For the remaining candidates, we repeat this entire process of extrapolation and matching, but only with the tracks that are on the list inside the objects and using the "second" extrapolator instead of the first one. That is, for each MuonTrackCandidates object, we make a new MuonTrackCandidates, loop over all tracks in the list from the original object, and redo the extrapolation with the second, and presumably more precise, extrapolator. (In practice, the same extrapolator is used for both passes.) We repeat the procedure of the previous paragraph, so that in the end the new MuonTrackCandidates objects have a set of better-extrapolated tracks which may be good matches to the stub. Once again, we remove candidate objects that have no tracks after this second pass.

Then it is time to determine the "best" track-stub matches. For each MuonTrackCandidates object, we sort the internal list of possible track matches in order of increasing value of q, *i.e.* in decreasing order of match quality. Then, the candidate objects are sorted in order of increasing value of the smallest value of q held within each candidate (follow that?). As a result, the track at the top of the list of tracks that is held inside the candidate that is in turn at the top of the list of candidates is the track that has the best match to any stub found in the event. This track-stub pair is deemed to be a muon, and a CdfMuon object is created for it. The MuonMatchData for this track-stub pair is also added to the muon. If the MuonTrackCandidates object that held the winning track-stub pair has another track that was a potential match to the stub, that track is chosen as the "second-best" track for the muon. (If there are multiple tracks available, then the one with the smallest q is chosen, of course.)

We then attempt to assign other stubs to this track. We look through the remaining candidate objects to see if the same track has been associated with any other stubs that are not in the same system as the original stub. If only one ther candidate object has that track, then the stub is simply added to the muon. If more than one stub in a new system has the track associated with it, we use the one that has the smallest q value.

Once this has been done, the resulting CdfMuon is added to the output list. The track is noted in a list of used tracks, and the MuonTrackCandidates objects containing the used stubs are taken out of circulation. Then, the arbitration phase is repeated. The best track-stub match of the remaining stubs is made into a muon, and other stubs are added to it. This continues until the list of stubs (or the reader) has been exhausted.

5.3.5 Calorimeter Quantites and Stubless Muons

Once all the track-stub associations have been made and arbitrated, the resulting CdfMuons are returned to the MuonLinkerModule. Here, the calorimeter quantities are associated with each muon. This work is done by the MuonCalorCalculator class.

The calculator takes the muon and the event-record pointer as inputs. (It can also take a CdfTrack as an input instead of a muon, so that a user can get the calorimeter quantities for an arbitrary track. This can be used to see if a charged particle is minimum ionizing.) It uses the event-record pointer to make a PhysicsTowerData object. This contains towers with transverse-energy values, using the z_0 of the muon track as the vertex to determine $\sin\theta$.

Then, the calculator uses the MuonExtrapTool to extrapolate the muon's track to four different locations – the entrance and exit points of the track for the electromagnetic and hadronic calorimeter. In each case, the track is extrapolated to the radius of the appropriate cylinder. If the track ends up being beyond the end of the calorimeter barrel in z, it is re-extrapolated to the plane of the plug. From these four points in space, the calculator determines which electromagnetic and hadronic towers have been traversed by the muon. The energies in those towers are summed to determine the electomagnetic and hadronic energy depositions of the muon, and those values and the `TowerKey` indices of the traversed towers are written into the `CdfMuon`. Neighbor-tower energy sums are also calculated, but not written into the muon, as the definition of "neighbor towers" depends so strongly on the muon's travel path.

Isolation sums are determined in the same way as is done in the electron reconstruction, so that muons and electrons can be treated the same way. The muon track is extrapolated to the CES, using the same tools as the electron code. The calculator creates `PhysicsTowerViews` for towers falling in cones of $\eta - \phi$ radius of 0.4 and 0.7 about that location. The energies in those towers are summed, excluding the towers that were traversed by the muon. The resulting sums are written into the `CdfMuon`.

At this point, some of the high-p_T tracks in the event have been successfully associated with stubs and made into `CdfMuons`, while others have not. The unassociated high-p_T tracks are made into `CdfMuons` without stubs at this point. (Here, high-p_T means greater than 10 GeV/c; this parameter is adjustable *via* a `MuonLinkerModule` talk-to.) These "muons" are run through the `MuonCalorCalculator` also, so that their calorimeter quantities are derived in the same way as the muons with stubs.

To finish the process of track-stub linking, the output `CdfMuon` objects (both with and without stubs) are sorted in order of decreasing track p_T, and written into the event record in a `CdfMuonColl`, with the description string `ProductionMuons` attached.

6 Tools

6.1 Muon Fiducial Tool

Frequently we will have a track reconstructed in the COT and/or SVX and will want to know where the track is headed after it leaves the COT. Will the trajectory hit one of the muon detectors? The muon fiducial tool answers this question. Not only will it tell whether the trajectory hits a muon detector, but it will also tell how close the trajectory comes to the muon detector.

The muon fidicial tool is not run as part of production; one has to include calls to it in one's own analysis code. The header file is `Muon/Muon/MuonFiducialTool.hh`. There is also a source file `Muon/src/MuonFiducialTool.cc`. Most of the code is in those two files, although the fiducial tool relies on the simple extrapolator and the standard CDF geometry structure as well.

The fiducial tool includes many functions, but there are two that are simple and will serve the needs of most users. They are used as follows:

```
MuonFiducialTool tool;
bool fiducial = tool.isFiducial(track, MuonDigiCode::CMX);
```

or

```
double distance;
MuonFiducialTool tool;
bool fiducial = tool.isFiducial(track, MuonDigiCode::CMX, distance);
```

The first two arguments are input arguments, a `CdfTrack` and an `enum` specifying the subdetector of interest (CMU, CMP, CMX, or BMU).

The return value will be true if the track is headed in a direction that will lead it to hit enough chambers for a stub to be reconstructed. All 4 subdetectors require 3 hits in 3 different layers for a stub to be reconstructed. The CMX also requires that two of those hits be on a seed pair (two chambers at the same phi).

The value returned in "distance" is signed, negative for fiducial muons and positive for non-fiducial ones. The absolute value of the distance is never larger than 50 cm. The algorithm stops looking at that distance and returns 50 for ones with a distance greater than 50 or returns -50 for ones with a distance less than -50. The distance returned is measured in the local x-z plane of the chambers near the place the track hits the muon detector. It is the distance the track must be translated to change from fiducial to non-fiducial or vice versa. In calculating this distance, changes in the direction of the track are not considered, only translations in the local x-z plane. The algorithm looks for the nearest such point: looking in the x direction, the z direction, *and* in all other directions that lie in the x-z plane. It returns only the distance of the nearest such point, not the size or shape of the fiducial/non-fiducial region the point lies in. One should take this into account when using this tool. Errors on the returned distance are about 10% or 2 mm, whichever is larger. This error estimate excludes errors associated with the geometry (built by code in the `MuonGeometry` package), and the extrapolator (done by code in the `Extrapolator` package).

This code was tested and designed for tracks passing the following cuts: $p_T > 1.3$ GeV, $|z_0| < 200$ cm, and $|d_0| < 25$ cm. It is highly recommended the user select tracks passing these cuts before passing them to the functions above. At some unknown point, the algorithm will fail if the track parameters fall outside this region. It probably will not crash, just return nonsense.

The calculations involved are a little slow, using significant CPU time. A very cursory benchmark was done using an executable compiled unoptimized. If the distance is requested, roughly 4 ms is used per track. Significantly less CPU time is used if the distance is not requested. It will save CPU time if this tool is only used on interesting tracks (*i.e.* apply some selection cuts first).

A track that is initially headed for a muon detector may multiple scatter and miss the detector; the probability of this depends on the track momentum. This fiducial tool does *not* consider multiple scattering. Users must appropriately take multiple scattering into account in their analysis on their own. This tool also does not consider things like chamber efficiencies or cuts made in the pattern recognition on things like stub slope.

The geometry comes from the standard CDF geometry structure. The tool determines if a track will hit a specific chamber by using the transformation matrix to local chamber coordinates for each chamber that is "near" the trajectory. These transformation matrices are the ones from the `CdfTangible` objects that have been corrected using the run-by-run position constants in the database (the proper database tables must be available or you will silently get the nominal geometry). The distance calculation algorithm is a search pattern where the track is translated to all points where it might change from being fiducial to non-fiducial or vice-versa. The fiducial calculation is repeated at each point.

The executable using this class must include the standard CDF geometry or it will not work. In addition, the detector of interest (CMU, CMP, CMX, or BMU) must have been built in the `MuonDetectorNode` in the geometry structure. If it is not, then all tracks will be non-fiducial. (For example, if BMU is not built into the geometry, all tracks will be non-fiducial in BMU). This will not be a problem if you use the default settings for the geometry.

The extrapolation of the track to the vicinity of each muon detector is performed by the `MExtrapolator` (simple extrapolator). Currently, this is hard-coded as the only option, although it should be possible to modify the code to use the `GEANT` extrapolator or add a switch to allow users to choose an extrapolator at run time. After the extrapolation to a position near the subdetector is done, all the short extrapolations to individual chambers are handled internally. These short internal extrapolations are all in a straight line. If one tries to use `GEANT`, one should think about tracks that range out (muons that are absorbed in the steel before reaching the muon detector of interest), as the tool may have difficulty handling them. In the current simple extrapolator, tracks never range out.

Internally, there is a function named `isAlive(digiCode)`. The argument is a `MuonDigiCode`. The rest of the functions in this class query `isAlive(...)` to determine which chambers are alive or dead. It is currently implemented using a hard-coded list of the chambers that exist. If this one function were modified to return false for dead/hot channels based on a database, then the rest of the code would properly take them into account when determining whether a track is fiducial and calculating the distance. Maybe someone will improve the `isAlive` function in the future to make database queries. One caution to anyone trying to do this: the function is heavily used by the algorithm so whatever implementation is used it must be fast (probably you want to cache the dead chamber list in an array).

For CMU, CMP, and BMU, the effects of the small gaps between adjacent chambers are not interesting for most purposes, because the gaps are smaller than the uncertainty in the extrapolated positions caused by multiple scattering. In the algorithm, small CMU/CMP/BMU gaps are treated as fiducial regions. A track that hits a gap between two chambers in the same layer less than 1 cm wide is considered fiducial in that layer. These gaps are also considered as fiducial regions when the distances are calculated. If the track hits a gap larger than 1 cm, it is considered non-fiducial.

In Run 1, it was possible to get the components of the distance in the local x and z directions as well as a distance. This functionality is provided.

```
double distance, xDistance, zDistance;
MuonFiducialTool tool;
bool fiducial = tool.isFiducial(track, MuonDigiCode::CMX,
                                distance,
                                xDistance,
                                zDistance);
```

The input arguments, return value, and the value returned in `distance` are exactly the same as in the two functions described above. The two new output arguments return the values of the x and z fiducial distances separately.

As described above, the local x axis and local z axis are exactly perpendicular to the faces of the chamber. The local x axis points in the direction closest to parallel to the drift direction. The local z axis is parallel to the drift wires.

The value returned in `xDistance` is a distance measured only in the local x direction. To calculate `xDistance`, one first makes the assumption that the drift cells are infinitely long, and then `xDistance` is the distance a fiducial track would have to be translated to become non-fiducial or the distance a non-fiducial track would have to be translated to become fiducial. (Here, we mean fiducial or non-fiducial in the infinitely-long chambers; below we use "fiducial in x" to mean the same thing). The `xDistance` is signed so that it is positive for tracks that are non-fiducial in x and negative for tracks that are fiducial in x. +50 cm will mean more than 50 cm *or* that the extrapolated track is very far from the detector (it could be very far away in the z direction, `xDistance` is not even calculated in this case). -50 cm will mean less than -50 cm.

The value returned in `zDistance` is the distance a track would have to be translated in the z direction to become fiducial if it is non-fiducial or vice versa. It is also signed, with positive values for non-fiducial tracks and negative values for fiducial ones. If the track would be fiducial if the cells were infinitely long, then this distance is measured at the local x of the extrapolated track. Otherwise, it is measured after the shortest translation in the local x direction that would make the track fiducial in x. A value of 50 cm means either the distance was greater than 50 cm or there were no chambers close enough to measure the distance in the z direction (`xDistance` will be 50 cm also in the second case). -50 cm means less than -50 cm.

There are some extensive comments at the top of the header file (MuonFiducialTool.hh). These comments go into more technical detail than is discussed here. Not only does header describe the technical details, but it also describes several other functions that are available in the class. For example, one can get internal information generated in the fiducial calculation. There are accessors that will return the MuonDigiCode of the nearest chamber or the position after the extrapolation or how many chambers were hit. There are also functions that allow one to use a CdfMuon instead of a CdfTrack as an input argument (or position, direction and charge). There also some options that might help one study the small gaps in CMU, CMP, and BMU.

A Re-running Muon Reconstruction

The muon-reconstruction code continues to improve over time. If you want to remake the muons in a given file, without changing anything else in the event, use a script that looks something like this. Your executable must have the MuonRecoSequence linked in.

```
#Create a path that includes the muon-reconstruction modules.  We
#don't want to redo the tracking, just the muons.
path create mypath ManagerSequence MuonRecoSequence
#You might want to drop all of the old muon objects from the event:
talk DHInput
  input file my.file #whatever this may be
  dropList set MuonXZHitColl MuonXHitColl MuonTHitColl MuonStubColl CdfMuonColl
exit
#By default, the muon code will remake all of these objects "from scratch,"
#i.e. starting from the D banks.  But you need to tell the MuonLinkerModule
#to use the old tracks
talk MuonLinkerModule
  findTrackProcessName set true
  trackProcessName set PROD #or whatever name is associated with your tracks
exit
#In your output, be sure to keep the new objects
talk FileOutput
  output create mystream new.file
  output path mystream mypath
  output keepList mystream -add MuonXZHitColl MuonXHitColl MuonTHitColl
  output keepList mystream -add MuonStubColl CdfMuonColl
exit
```

B Using the G3X Extrapolator

As described in the text, GEANT-based extrapolator, referred to here as G3X, can be used to estimate track-stub matching χ^2 values for all muons. GEANT is a powerful and effective program that has been developed over many years. However, it has also grown to be a little large. The offline analysis managers have decided that it should not be run as part of the standard production executable. If you wish to use G3X, you'll have to re-make the muons yourself.

To do so, you'll have to build a special version of the reconstruction program. The simplest way to do this is to use the MuonMods_Test executable that is part of the MuonMods package, which does have the MuonRecoSequence linked in. Check out the package into your test release, set the environment variable USE_G3X to true, and build the package in the normal way. Then, run MuonMods_Test with an script that looks something like this:

```
#You won't be redoing tracking, so save yourself time fetching constants
module disable CT_TrackingModule
#Make a path
path create mypath ManagerSequence SimInitManager MuonLinkerModule
path enable mypath

#Set up the linker to use tracks and stubs that already exist in your
#input file.  If PROD is not the process that created them, change that word.
talk MuonLinkerModule
findProcessName set t
processName set PROD
findTrackProcessName set t
trackProcessName set PROD
#This is the crucial line -- don't include it if you don't want GEANT!
secondExtrapolator set G3X
exit

#Turn on the GEANT initialization.  Sorry about this, still agitating to get
#defaults set correctly.
#but doing this talk-to seems necessary.
module enable SimInitManager
talk SimInitManager
  DetectorMenu
     declareAll set t
     declareCpr set f
  exit
exit

#Input/output stuff
talk DHInput
#You might want to save the old muons, or do this if you don't want
#to get confused
dropList set CdfMuonColl
input file /mydisk/myfile.dat
exit
talk DHOutput
output create mystream /mydisk/mynewfile.dat
output path mystream mypath
exit
```

The muons in the output file will have all of their χ^2 values filled. Note that the various χ^2 quantities are not guaranteed to have true χ^2 distributions, due to tails in the matching distributions that are not understood.

References

[1] The CDF II Detector: Technical Design Report, November 1996.

[2] See, for instance, CDF 6029 and CDF 6015.

[3] CDF 4152, "The CDF Run II Event Structure."

[4] CDF 4816, John Marriner, "Muon Upgrade Tube Drift Model," May 1992.

[5] CDF 5715, Kevin Lannon, "Everything I Know About CMU Charge Division, I Learned From Lee."

[6] CDF 2871, David Reich *et al.*, "A New Algorithm for Stub Reconstruction in the CMX," 26 January 1995.

[7] CDF 5079, Patrick Lukens, "The CMX Lower Wire Positions," 16 August 1999.

[8] CDF 5055, Daniel Larson *et al.*, "CMX Wire Positions: Arches," 16 July 1999.

[9] CDF 5815, John Marriner, "The CMEX Stub Finding Description," 23 March 1995.

[10] CDF 1430, D. Frei, "Multiple Scattering of Central Muons," 1991.

[11] R. Brun, F. Carminati, "GEANT Detector Description and Simulation Tool," CERN Program Library Long Write-up W5013, see also `http://consult.cern.ch/writeup/geant/`.