

The Connection Between Hits and Geometry

Rob Kutschke, Fermilab CD

Abstract

This note presents a sketch of how one might connect tracking hit information with its associated geometry information, where “tracking” includes both pixels and strips. The note is intended to as a way point on the discussion we started a few weeks ago and will include: a sketch of how hit information flows, in experimental data, from the detector to its incorporation into tracks; the corresponding path for hits from Monte Carlo events; and a discussion of matching Monte Carlo reconstructed tracks with generated tracks.

Contents

1	Introduction	1
2	Local and Global Coordinate Systems	2
3	Data Flow for Experimental Data I	3
3.1	Packed Binary Data	3
3.2	Unpacked Raw Data	3
3.3	Connecting the Hits to the Geometry	5
3.4	Clusters of Hits	7
3.5	Crosses of Strips	8
3.6	Derived Data Products	8
4	Data Flow for MC Hits I	9
4.1	Precursors	9
A	Information Needed in SensorGeom	10
B	Some Further Comments on the Geometry Subsystem	12
C	Questions	13

1 Introduction

For track finding and fitting, the main geometrical unit is the sensor. On pages 8 and 9 of the report we presented at Beijing, I count 8228 sensors in the barrel

silicon strip tracker. As a rough guess I will multiply this by 3 to include sensors in the forward tracker plus the sensors in the pixel system. In round numbers we have of order 25,000 sensors in the entire system.

In the following examples I will show some code fragments. For now they are written in “mock C++” just because that is the way I think. Hopefully this is clear enough for this point in the discussion. Also, I have paid no attention to the uses of inheritance and polymorphism; I think we need first to decide what we want and then to decide where inheritance and polymorphism improve the design.

Given a hit, we need fast, easy access to the geometry information. I propose that the way to do this is:

1. To assign each wafer a unique Id.
2. For the geometry system to support a function that returns sensor geometry information given the sensor Id.

The corresponding code fragment is,

```
class SensorId;
class SensorGeom;
SensorGeom& getSensorGeom( SensorId );
```

where `SensorGeom` is the information we need and where the function `getSensorGeom` lives within the appropriate scope within the geometry system. An illustration of the information that must be provided by `SensorGeom` is in Appendix A.

My guess is that `SensorId` will just be a dense integer, so that one can use it as an index into indirection arrays. But for now it’s just a class.

I have tried to choose class names that do not match any of the existing `org.lcsim` or `SLIC` classes. The intention is to give us nomenclature to separately discuss the classes we need and the classes we have. Later we can discuss how to morph the classes we have into those that we need.

2 Local and Global Coordinate Systems

In his recent talk, Rich defined the (u, v, w) notation to describe local coordinates on a particular sensor. In this note (x, y, z) are always global coordinates and (u, v, w) are always local coordinates. As a reminder, for a strip sensor, the local coordinates are define by one vector and three unit vectors,

- \vec{r}_0 vector to the local origin on the sensor.
- \hat{u} in the plane of the sensor, along the measurement direction.
- \hat{w} normal to the plane.
- \hat{v} $\hat{w} \times \hat{u}$, in the plane of the sensor, along the strip direction.

These vectors are expressed in global coordinates; that is, \vec{r}_0 is the location, in global coordinates, of the local origin of a sensor, and \hat{u} is the measurement

direction on some sensor plane, expressed in global coordinates. Note that, for an ideally aligned barrel strip sensor, \hat{v} is along $\pm\hat{z}$ and, for an ideally aligned forward sensor, \hat{w} is along $\pm\hat{z}$.

For a pixel sensor, \hat{u} and \hat{v} are the two orthogonal measurement directions in the plane.

3 Data Flow for Experimental Data I

This section discusses data flow from the detector to the tracking algorithms. It will illustrate how the geometry information can be connected to the hits in a way that I think is economical of both memory and CPU time. For simplicity I will only consider barrel strip detectors but I will presume that we want to consider some barrel layers which have two sensor layers, the second containing some sort of stereo information. The data flow is illustrated in Figure 1 and the corresponding classes are discussed below.

3.1 Packed Binary Data

The rawest form of data is packed binary data straight from the event builder. As far as this note is concerned, it is just a blob that can be unpacked. We do not need to know the internal structure. This is represented by the top box in Figure 1. I presume that this data is held by the event data model (EDM).

3.2 Unpacked Raw Data

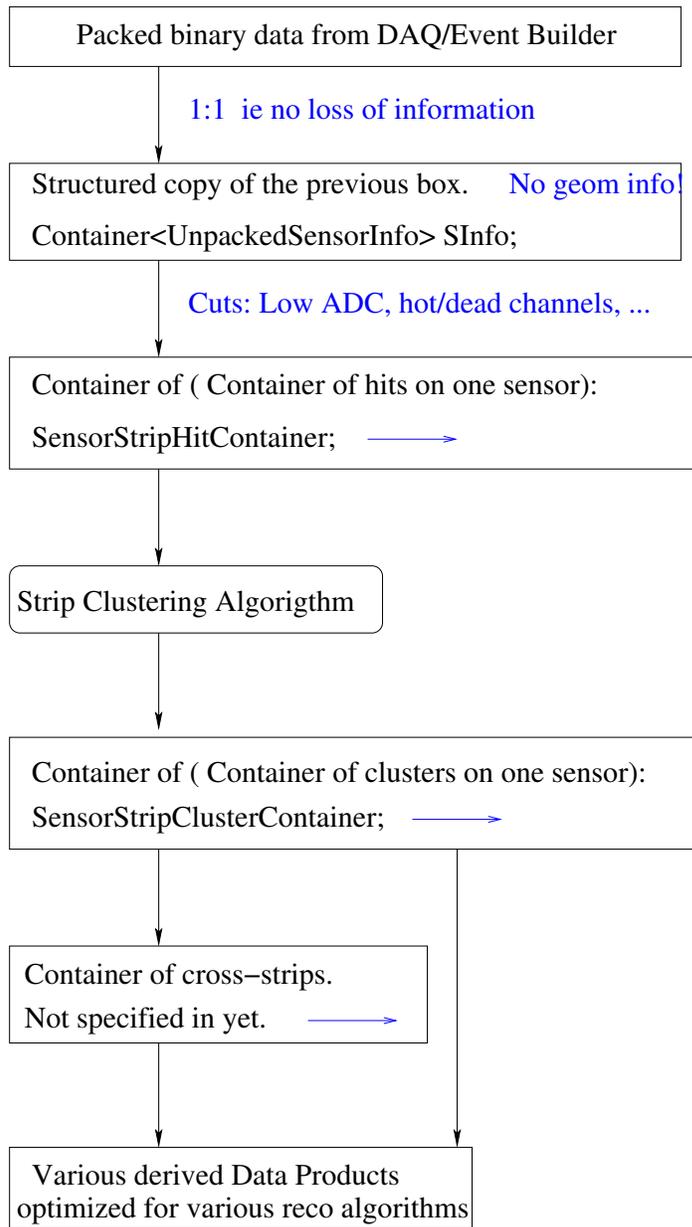
The first step in processing the data is to unpack it in order to provide structured access to the information. This process should be lossless and it should report errors when it encounters inconsistent data. This produces the second box from the top in Figure 1. This box holds a single container of objects called `UnpackedSensorInfo`, each of which represents a single hit strip:

```
class ElectronicAddress{
    // Not specified in detail.
    // It contains a chip Id, a channel number within the chip ...
};

class TimeStamp{
    // Not defined in detail.
};

class UnpackedSensorInfo{

public:
    // Data members
    ElectronicAddress add;
    short int iadc;
```



→ Denotes objects that have references to geometry object

Figure 1: The main objects on the path of barrel silicon strip data as it starts at the detector and is processed into tracks. The items in rectangles represent objects that are held by the event data model (EDM). Outlines of the objects are given in the text.

```
    TimeStamp t;
};
```

By design these classes are independent of the geometry and calibration systems. They are simply a structured representation of the data at the previous level.

I am not sure that we need a timestamp but I have included one for completeness.

Each `UnpackedSensorInfo` object is really just a struct whose data is complete at instantiation and never changes until the object is destructed. These objects should be treated as such; that is their only methods are their constructors and destructors. We can debate another day if we should make the data private and provide accessor functions.

I presume that the `Container<UnpackedSensorInfo>` object is held by the EDM.

3.3 Connecting the Hits to the Geometry

The third box in in Figure 1 contains the first objects in which the hit information is connected to the geometry and the conditions data. This box holds a container of containers of hit strips, with one outer container per sensor. The object representing a single hit strip is,

```
typedef int StripNumber;
typedef double ADC;

class StripHit{
public:
    StripNumber iu;
    short int iadc;
    ADC adc;
    TimeStamp t;
};
```

These objects are collected into containers, one per sensor,

```
class SensorStripHits{
public:
    SensorId sid;
    SensorGeom& sgeom;
    Container<StripHit> hit;
};
```

where I have used “Container” as a placeholder for your favorite container type. The object that holds all of the hits for all of the sensors will look something like,

```
class SensorStripHitContainer{
```

```

// Accessor methods:
public:
    bool                hasStripHits( SensorId );
    SensorStripHits& getStripHits( SensorId );

private:
    map<SensorId,SensorStripHits> SSHits;
};

```

For conceptual purposes, I have represented the big container of containers as map. On the other hand, if `SensorId` is just a dense integer, this could be implemented an array of `SensorStripHits`:

```
vector<SensorStripHits> SSHits[SensorId];
```

This is would yield fast retrieval of the geometry information given a `SensorId`.

I presume that the `SensorStripHitContainer` object is held by the EDM. Some additional comments on this design,

1. The class `SensorStripHits` contains a reference to the geometry of the sensor. This is the main point of this document: it is one way to ensure that, whenever you have access to some hits, you also have fast access to the associated geometry information.
2. This reference is computed only once per sensor per event.
3. The grouping of hits by sensor also connects this class with the geometry.
4. The class `SensorStripHits` contains a copy of the `SensorId`. This is might be redundant but it also might be useful for debugging.

The transformation from the previous box, in its full complexity, is a heavy-weight step:

1. It needs access to the full conditions data base to access pedestals and gains for the ADCs, to access the list of hot/dead channels, and to access the map from electronic addresses to sensor/strip.
2. It is allowed to discard data. One might discard data because the ADC is below threshold, because the strip is on a list of known hot/dead channels or because the hit is out of time.
3. There no links from this container back to precursor containers: the only repeated data are `iadc` and the timestamp, which are about the same impact as a back link.

3.4 Clusters of Hits

The next step in the data processing is to form clusters of hit strips, which is represented by the fifth box in Figure 1. A cluster may contain a one or more strips. Cluster finding takes place within each sensor, without reference to other sensors, so the organization of the preceding section can be duplicated.

```
class StripCluster{
    // Data Members
    Container<int> strips;
    double pulseheight;
    double centroid;           // Measures u
    double sigma;             // Error on u.

    // Member functions
    double Centriod( double theta);
    double Sigma( double theta);
    double Pulseheight( double theta );
};
```

The first data member is a list of the strips that make up the cluster. In this example it is implemented as a container of ints; each int is an index into the `SensorStripHits` container for this sensor. We could also choose to implement this as an array of references to those hits. The next data member is the pulseheight of the cluster, the sum of the pulseheights of the contributing strips. The next two data members are the first guess at the centroid and the error on the centroid, which are made without associating the cluster with any track. Both of these are measurements along the \hat{u} direction. I chose to drop the timestamp in this object; if needed, it can be found by reference to the contributing strips.

There are three member functions, which can only be used once the cluster is associated with a track. All three take an argument which is the angle that the track makes with the normal to the sensor. One can get an improved estimate of the centroid and the sigma when this angle is known. I am not sure if we learn anything more about the pulse height once the angle of incidence is known but I have included such as function for completeness.

The container that holds all of the clusters on a given sensor is modelled on `SensorStripHits` container; again it contains a connection to the geometry.

```
class SensorStripClusters{
public:
    SensorId sid;
    SensorGeom& sgeom;
    Container<StripCluster> hit;
};
```

Again there needs to be an object to hold the container of containers,

```

class SensorStripClusterContainer{

    // Accessor methods:
    public:
        bool                hasStripClusters( SensorId );
        SensorStripClusters& getStripClusters( SensorId );

    private:
        map<SensorId,SensorStripClusters> SSClusters;
};

```

Again the use of map is just for conceptual purposes and other implementations are possible, perhaps preferred.

I presume that the `SensorStripClusterContainer` object is held by the EDM.

3.5 Crosses of Strips

In a traditional double sided strip detector, the readout strips on the back side run orthogonal to those on the front side. That is, if the strips on the front side measure the \hat{u} coordinate, then those on the back measure the \hat{v} coordinate. There are no such sensors planned for SiD but the options under consideration do include crossed single sided sensors placed more or less back to back.

For a traditional double sided sensor, the next step in data processing is to form “crosses”, objects that describe the (u, v) intersection point of each pair of strips. Consider, for example, a sensor that has two real tracks pass through it. In the typical case this will create two clusters of hit strips on each side, which can be paired to make four crosses, two which correspond to real track intersection points and two of which are ghosts. In my experience the crosses were the objects used by pattern recognition code but the fitters used directly the underlying strip information.

In the designs discussed for SiD we probably need a more general solution than this. In the forward tracker we are talking about crossed one-sided strips. However the crossing angle is not necessarily 90 degrees. Even if it were designed to be 90 degrees, construction tolerances would make it something else. More the two sensor planes are not necessarily back to back, which means that a “u-measuring” sensor might overlap two different “v-measuring” sensors.

Therefore we will leave the classes related to crosses, and containers of crosses, unspecified for now. We just note that they will be needed and that the big container of containers will be held by the EDM.

3.6 Derived Data Products

The bottom box in Figure 1 is there to remind us that particular pattern recognition and fitting algorithms will need derived data products. These data products will generally take two forms:

- Different organizations of the same information, organized to provide fast convenient access for some algorithm.
- Once you connect a track to a cluster , you can compute a new estimate of the centroid. During the pattern recognition process, one may try one cluster on many different candidate tracks. Therefore it may be useful to create many hits-on-a-track objects from one cluster object, each hit-on-a-track attached to a different candidate track.

4 Data Flow for MC Hits I

This section is the analog of the previous section but for hits that originate in MC simulation, not from the detector. Again, for simplicity we will only consider barrel strip detectors. Figure 2 shows the an outline of data flow; it differs from the earlier figure only in the first two boxes, drawn in red.

4.1 Precursors

The precursors are not described in detail in this note. They include the `SimTrackHit` objects and any intermediate objects used to compute the pulse-height on each strip. The class that describes the pulse height on each strip will look like,

```
class MCTrackId{
    // Not specified yet.
};

typedef double ExactADC;

class MCTrackADC{
    MCTrackId  trk;
    ExactADC   adc;
    Timestamp  t;
};
```

In this picture, we need some sort of identifier, `MCTrackID`, to uniquely identify every track in the MC chain. Given just this identifier it should be possible to trace where the particle comes from. Among other things, `MCTrackId` needs to be rich enough that it can identify tracks and photons that come from beamstrahlung. We also need to define the idea of the correct known contribution to the pulseheight, `ExactADC`; this includes the effects of diffusion, recombination, Lorentz forces, and the Hall effect but it does not include the response function of the electronics. Next we need a class to tell us that a given track created a certain pulse height on a given strip and that it did so at some time, `MCTrackAdC`.

With the above tools we can construct a class to summarize the full history of the pulse height on a each strip,

```

class MCStripHit{
public:
    StripNumber iu;
    Container<MCTrackADC> Energy;
    ExactADC noise;
    ExactADC adcsum;
    Timestamp t;
};

```

I propose to collect these in a container of containers, exactly parallel to the container structure for hits from the experiment:

Presumably this is all persistable but we would only persist it when doing detailed studies. For physics benchmarking studies we would drop it.

Glitch with hits below threshold in MC? No: there is a separate threshold for clustering?

A Information Needed in SensorGeom

The following class illustrates the information needed from the geometry system for each sensor.

```

class SensorGeom{
public:
    SensorID id;
    SubSystem type;           // See below. Might not be needed?
    double r[3];              //  $\hat{r}_0$ .
    double u[3], v[3], w[3]; // unit vectors along the u,v,w axes.
    double dim[3];           // Dimensions along the u,v,w axes.
    double pitch[2];         // Pitch along the u and v axes.

                                // Do we need?
    ?????                    // A reference back to full geom info?
};

```

where SubSystem is a bookkeeping convenience defined by,

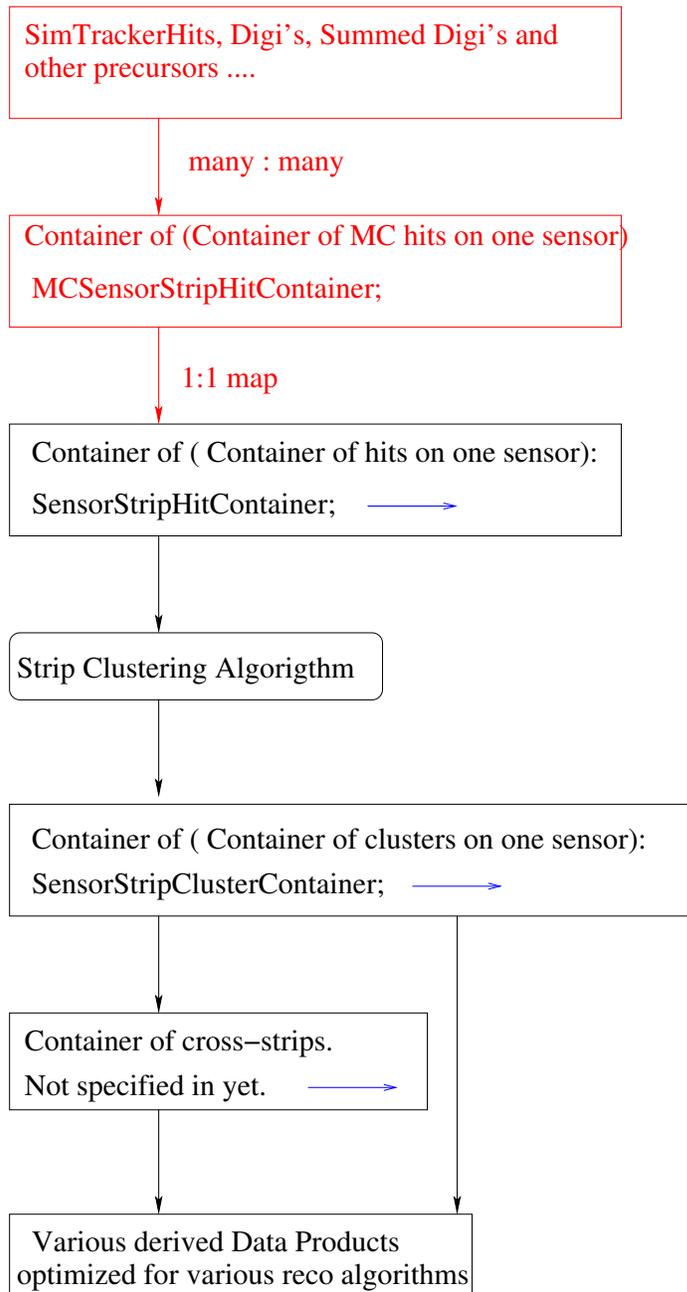
```

enum SubSystem { BarrelPixel, ForwardPixel,
                BarrelStrip, ForwardStrip};
int nSubSystems(ForwardStrip+1);

```

If the equivalent information is already directly, *and quickly*, available, from the existing geometry system, we do not need this class and our code can talk directly to the existing geometry system.

It is redundant to carry the SensorId around in the object but I find it a powerful debugging convenience.



→ Denotes objects that have references to geometry object

Figure 2: The main objects on the path of barrel silicon strip data as it starts at in MC simulation and is processed into tracks. The items in rectangles represent objects that are held by the event data model (EDM). Outlines of the objects are given in the text.

I have not defined any methods for this class since it is essentially a convenience class for accessing a subset of the information on each sensor. If someone wants the full power of the geometry system, they can follow the pointer/reference back to the geometry system and access its methods directly.

To complete the geometry specification we need to define a few conventions, such as: Conventions we need to define to complete this:

- Where is the local origin: at the center of the “top” surface, at the body center, at a corner, at an edge?
- Are dimension full lengths or half lengths?
- Is \hat{w} an inward normal or outward normal or someother convention.
- For pixels, which is u and which is v?
- Do we want to define now the methods one would need to deal with non-flat wafers?

B Some Further Comments on the Geometry Subsystem

Here are some thoughts about some useful methods that might live within the geometry system.

Suppose that I have a candidate track that has hits on some subset of the layers and I want to look to the next layer and ask if there are any hits from that layer that fit the track candidate. Here is how I think of the problem.

1. Extrapolate my track to the nominal radius of the next barrel layer, or the nominal z of the next forward layer. Probably this is done using the full knowledge of the magnetic field.
2. Look for sensors in that layer that fall within some “search window” around the extrapolated track. Return a list of `SensorId`'s for all sensors that fall within the window. Because sensors have overlaps, it is always possible to return more than one sensor, even with an arbitrarily small search window. It might not be necessary to consider the magnetic field when doing this search?
 - At later stages of the track finding/fitting, the search window might be defined by the footprint (covariance matrix) of the extrapolated track. At earlier stages of track finding the, the search window might be some nominal distance that is a parameter of the algorithm.
3. For each sensor in the list, get the `SensorStripHits` container and test hits in that container to see if they are compatible with the track.

This use case suggests that the geometry subsystem (or some system) needs a function like:

```
Container<SensorId> FindIntersections ( SubSystem, Layer, Track&, SearchWindow );
```

where SubSystem was defined earlier, and Layer is a layer number within that subsystem, where Track is an object that holds track parameters plus a covariance matrix, and where the search window is somehow specified by the last parameter. In an actual implementation, the arguments might really be references to objects within the geometry system rather than codes/indices into the geometry system that are present in this example.

One could also imagine a more general function which finds the next group of sensors encountered by a track, considering all layers in all subsystems.

```
Container<SensorId> FindIntersections ( Track&, SearchWindow );
```

In such a function the search window might need to be a function of arc length from the last measurement.

The latter function is probably more useful for a general prototype pattern recognition program while the former function is probably more useful for a highly tuned pattern recognition program that exploits knowledge of the detector in order to gain speed.

C Questions

1. I need to include Dima's suggestion about Clusters- \rightarrow Hits.
2. Hit is really badly overloaded. Fix it.
3. Raw is badly overloaded. Fix it.
4. How detailed is MCTrack Id. Do we want to trace every delta ray? What about albedo? Do we care about the full history of the albedo particle or just that it is albedo?