NuMI-NOTE-COMP-0860

Users Guide to the CandEventSR Reconstruction Tree Version 2.00

Roy Lee

Harvard University

April 18, 2003

Table of Contents

1 Introduction

This memo provides a description of the `reco` tree output by the method EventSRListModule::Ana. Each stage of reconstruction (e.g. track formation) can have its own set of trees being written out containing information about that package and upstream of it, and for many uses, such as analyses of cosmic muons, this is sufficient. However, for more complicated events, such as beam induced events in the near detector, a more complex tree structure complex is warranted, and this is the role of the CandEventSR reconstruction tree.

The file output by EventSRListModule::Ana containing the reconstruction tree is called `eventsr.root`. This ROOT file contains a single Tree with name `reco`. Information about CandStrip, CandSlice, CandTrack, CandShower, and CandEvent objects is contained within the `reco` tree. A description of these objects and of the MINOS reconstruction can be found in [1].

Throughout this memo there will be references to ROOT trees and their usage. Some basic knowledge of these trees by the user is assumed. For information on ROOT trees (and ROOT in general), the ROOT Users Guide is a good reference (available via http://root.cern.ch).

Note that as of the time that revision 2.00 of this note was being written, Sue Kasahara was in the process of converting the functionality found in the CandEventSR tree into the package CandNtupleSR. Eventually the code in EventSRListModule::Ana will become obsolete and users should use the new CandNtupleSR package when available.

2 Description

The `reco` tree is an example of a tree containing an object. In this case, the object being written is of class TTreeSR (code for all of the classes associated with the reconstruction tree is contained in the CandEventSR package). An examination of TTReeSR.h shows

```
class TTreeSR : public TObject
{
  ...
  EventSRHeader evthdr;
  CosmicRayInfoSR crhdr;
  VetoShieldInfoSR vetohdr;
  DmxStatusTree dmxstatus;
  TClonesArray *vetostp; // veto strip
  TClonesArray *stp; // strip
  TClonesArray *slc; // slice
  TClonesArray *trk; // track
  TClonesArray *shw; // shower
  TClonesArray *evt; // event
  TClonesArray *mc; //-> Monte Carlo truth
  TClonesArray *flsdgt; //-> Monte Carlo digit truth
};
```

There is one object of class EventSRHeader, one object of class CosmicRayInfoSR, one object of class VetoShieldInfoSR, one object of class DmxStatusTree, and several TClonesArray objects. The use of TClonesArray allows for multiple objects per tree entry, where there is a one to one relationship between a tree entry and a snarl. Strip objects obviously require an array, as there are multiple strips in a snarl. Most far detector snarls correspond to a single physics events, which means that the typical far detector snarl will have only a single track or single shower. However, this is not always the case, for example cosmic ray multiple muon events, or a ν_μ charged current interaction with muon and pion tracks. In the near detector this situation is even more complicated, where a near detector snarl can contain many physics events.

2.1 EventSRHeader

The class EventSRHeader contains information about each snarl:

```
class EventSRHeader : public TObject
{
  ...
  Int_t run;
  Short_t subrun;
  Short_t runtype;
  Int_t snarl;
  UInt_t trigsrc;
  UInt_t errorcode;
  Double_t trigtime;

  UInt_t ndigit;
  UInt_t nstrip;
  UShort_t nslice;
  UShort_t ntrack;
  UShort_t nshower;
  UShort_t nevent;
  DigitPulseHeightSR ph;
  PlaneInfoSR planeall; // all digits
  PlaneInfoSR plane;    // digits above threshold (nominally 3 pe)
// PlaneInfoSR plane requires 4 consecutive hit planes
  DateInfoSR date;
};
```

The first group of variables contain information identifying the snarl and are filled from the classes RawDaqHeader and RawDaqSnarlHeader, except for `trigtime`, which is calculated from the data itself if FilterDigitListModule has been run. The second group of variables show how many of each candidate objects are in the snarl.

The last group holds information about the total pulse height in the snarl (DigitPulseHeightSR), plane information (PlaneInfoSR), and date information (DateInfoSR). DigitPulseHeightSR has the structure

```
class DigitPulseHeightSR : public TObject
{
  ...
  Float_t raw;
  Float_t siglin;
  Float_t sigcor;
  Float_t pe;
};
```

where `raw` gives the pulse height in raw ADC counts, `siglin` returns pulse height corrected for nonlinearities, `sigcor` returns a normalized strip response pulse height, and `pe` gives the pulse height in photoelectrons. The summation is done over all CandDigit objects associated with the snarl.

There are two PlaneInfoSR objects, `planeall` and `plane`, which contain information about the beginning and ending planes as well as the total number of hit planes. This information exists both irrespective of the plane view, and for the U and V views separately. The first object, `planeall`, examines all CandDigit objects. This means that an upstream accidental hit can cause the beginning plane in planeall to be upstream of the actual physics event. An attempt is made to filter out such accidental hits when calculating the plane information. This is done by finding the most upstream plane which is part of a set of 4 contiguous planes, with each of the planes having at least 3 photoelectrons. This plane is defined to be `plane.beg` (similarly for `plane.end`). The total number of hit planes (without any pulse height requirement) between `plane.beg` and `plane.end` is then stored in `plane.n`.

Lastly, DateInfoSR holds information about the date of the snarl. All variables in this class are self explanatory, perhaps with the exception of the variable `utc`. This variable represents the universal time (in seconds) and is filled from the method VldTimeStamp::GetSec().

2.2 CosmicRayInfoSR

This class holds information that is particularly useful for doing analysis of cosmic ray muons:

```
class CosmicRayInfoSR : public TObject
{
  ...
  Float_t zenith;
  Float_t azimuth;
  Float_t ra;
  Float_t rahourangle;
  Float_t dec;
  Double_t juliandate;
  Float_t locsiderialtime;
};
```

These quantities are defined for reconstructed tracks only. For the case in which there is more than one track per snarl, these values are calculated for the last reconstructed track. When no tracks are reconstructed, these values are undefined.

The `zenith` and `azimuth` angles (in units of degrees) are defined from track direction cosines at the vertex. The `azimuth` is defined to be the true azimuthal angle; detector north as defined by the longitudinal axis of the far detector is offset by 26.4234474° with respect to true north. Note that these quantities are defined purely in terms of the far detector.

The other member variables describe time in astronomy terms, `ra` being the right ascension, `rahourangle` the right ascension hour angle, `dec` the declination, `juliandate` the julian date, and `locsiderialtime` the local siderial time. The Soudan mine sits at a longitude of -92.241389° (92-14'-31.23" W) and a latitude of 47.819722° (47-49'-13.29" N).

2.3 VetoShieldInfoSR

The far detector has a configuration of scintillator modules above and to the sides of the detector to identify muons which enter the detector. This veto shield is currently set up as a prototype, and it is more than likely that the permanent configuration will be different than the one that presently exists. Some of the member variables of this class may therefore need to be changed in the future.

The class VetoShieldInfoSR contains summary information about digits in the veto shield:

```
class VetoShieldInfoSR : public TObject
{
  ...
  UInt_t ndigit[3];
  UInt_t nplank[3];
  Int_t adc[3];
  Float_t dx[3];
  Int_t dxvetostp[3];
  Float_t dcos; // normal direction cosine to shield
  Float_t projx;
  Float_t projy;
  Float_t projz;
  Bool_t ishit;
}
```

Contiguous scintillator strips in the veto shield are grouped together and are read out by a single PMT pixel. We refer to these sets (mostly of 8 scintillator strips) as planks.

Most of the member variables are self explanatory. The reconstructed track (if it exists) is projected to the veto shield; the spatial residuals between the projected track position and the closest shield hits are calculated and stored in `dx`. The `vetostp` index of the veto shield digit whose spatial residual is smallest is represented by `dxvetostp`. The three elements of the arrays `ndigit`, `nplank`, `adc`, `dx`, and `dxvetostp` represent early, in time, and late shield digits, where the timing is determined relative to the track vertex time (if a track exists), and is corrected for time walk and optical fiber propagation delay. Early digits are nominally considered to be those which come at least 50 ns before the track vertex; late digits are nominally considered to be those which come at least 150 ns after the track vertex.

The projected track angle is calculated relative to the part of the veto shield which is intercepted, and the direction cosine relative to the shield is found and stored in `dcos`. The spatial coordinates of the projected track position are stored in `projx`, `projy`, and `projz`. Finally, the boolean `ishit` is true if the projected track intercepts the veto shield.

2.4 ShieldStripSR

The TClonesArray *vetostp is an array of shield strip objects, where a shield strip is defined to be a hit plank (see the previous section for definition of a plank) with either one or two digits. The structure of ShieldStripSR is

```
class ShieldStripSR : public TObject
{
  ...
  Int_t pln;
  Int_t plank;
  Float_t x;
  Float_t y;
  Float_t z[2];
  Int_t ndigit;
  Int_t adc[2]; // 0 = south, 1 = north
  Double_t time[2]; // 0 = south, 1 = north
  Double_t timeraw[2]; // 0 = south, 1 = north
  Float_t wlspigtail[2];
  Float_t clearlen[2];
}
```

The pln variable represents the plane number as defined by the Plex package. The plank is the strip number as defined by the Plex package of the first PlexStripEndId. The spatial position of the shield strip is found in x, y, and z, where the last variable contains the upstream and downstream ends of the shield scintillator strip.

The timing variables time and timeraw give the 3D fully corrected and raw times, where the event trigger time has been subtracted off. The 3D time correction includes a pulse height dependent time walk correction and corrections for optical fiber propagation delays. The south end is index 0, the north end is index 1.

2.5 StripSRTTree

The TClonesArray *stp contains information about CandStrip objects, with each element of *stp corresponding to a single CandStrip object. The structure of StripSRTTree is

```
class StripSRTTree : public TObject
{
  ...
  Int_t index;
  UShort_t strip;
  Float_t tpos;
  UShort_t plane;
  Float_t z;
  Char_t planeview;
  DigitPulseHeightSR ph0; // 0 = east, 1 = west
  DigitPulseHeightSR ph1; // 0 = east, 1 = west
  UShort_t ndigit;
  Float_t time0; // 0 = east, 1 = west
  Float_t time1; // 0 = east, 1 = west
  Int_t pmtindex0; // 0 = east, 1 = west
  Int_t pmtindex1; // 0 = east, 1 = west
};
```

All of these variables should be self explanatory. The 0 and 1 flags for `ph`, `time`, and `pmtindex` refer to the east and west read out ends (for the near detector, only one of these will be sensible).

The object `*stp`, being a TClonesArray, can be accessed using conventional a C style array index. For example, to print the transverse position, z position, and pulse height (in photoelectrons) for the 5th strip for snarl 69:

```
  reco->Scan("stp[4].tpos:stp[4].z:stp[4].ph.pe","evthdr.snarl==69");
```

To draw the transverse versus longitudinal positions for all strips in snarl 69:

```
  reco->Draw("stp[].tpos:stp[].z","evthdr.snarl==69");
```

The square brackets without any index tells ROOT to loop over all valid entries of the array. Alternatively, omitting the square brackets altogether means the same thing:

```
  reco->Draw("stp.tpos:stp.z","evthdr.snarl==69");
```

The result is shown in Fig. 1.

One word of warning: because writing out information about each CandStrip object can require a relatively large amount of disk space, there is an option in the EventSRListModule package to turn this off. If such is the case, `reco->Print()` will show the existence of the `*stp` array, but there will be no entries in it.

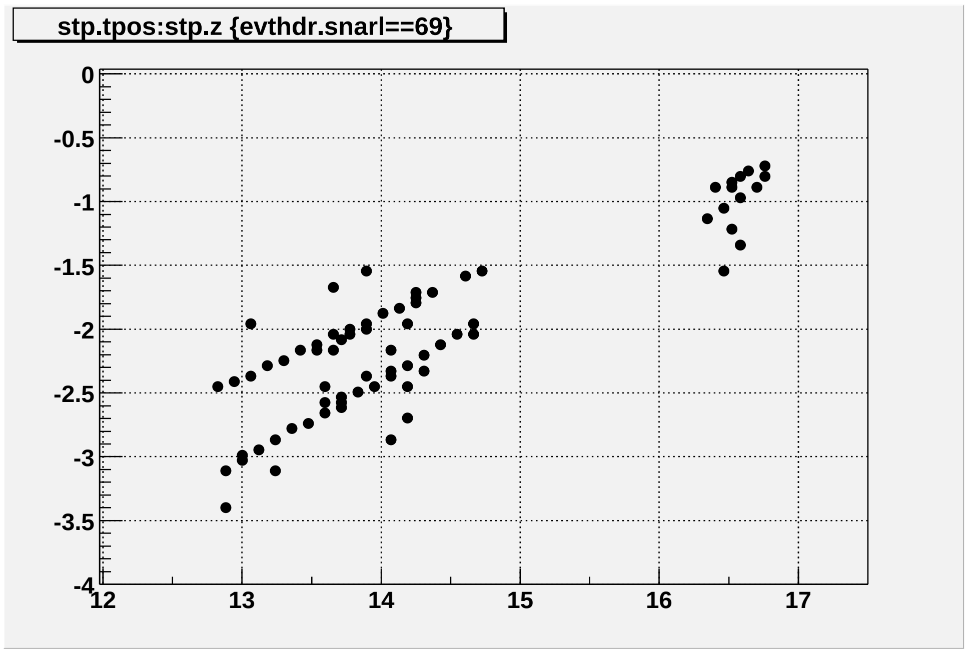

Figure 1: tpos vs z for all CandStrip objects

2.6 SliceSRTTree

A snarl may contain multiple physics events, particularly true in the case of near detector beam induced events. One of the first reconstruction tasks involves the formation of CandSlice objects. Such objects are formed by separating CandStrip objects based on timing and/or spatial information.

For far detector events, where a snarl contains a single physics event, there will be exactly one CandSlice object per snarl. This means that all CandStrip objects found in the snarl are also in the CandSlice, and vice versa. So while the use of CandSlice is of primary benefit to near detector beam events, some aspects of how to associate elements of the `*stp` array with a particular higher level reconstruction object (in this case a CandSlice) will be discussed here and may be beneficial to all users.

The SliceSRTTree class has the following members:

```
class SliceSRTTree : public TObject
{
  UShort_t index;
  Int_t ndigit;
  Int_t nstrip;
  Int_t *stp; //[nstrip]
  DigitPulseHeightSR ph;
  PlaneInfoSR plane;
};
```

The integer array `*stp` in SliceSRTTree is an index over the `*stp` array in TTreeSR. In the far detector where a snarl is equivalent to a CandSlice, the `*stp` array in SliceSRTTree contains all indices from 0 up to `nstrip`-1. Thus if one executed the command

```
  reco->Draw("slc.stp","","",1,0);
```

one would see a histogram from 0 to `nstrip`-1 with all entries being filled (the 1 near the end tells ROOT to consider only one entry, the 0 means begin with entry 0).

As `*slc` itself is an array (of StripSRTTree's), and `slc.*stp` is also an array of (integers), what is actually being drawn by the command above? We have not specified any array indices. What ROOT does in this case is to first loop over all elements of `*slc`, and for each element (of type StripSRTTree) loop over all elements of `slc.*stp`.

If one wanted to plot only those strip indices for the 4th CandSlice:

```
  reco->Draw("slc[3].stp","","",1,0);
```

Alternatively, the following command is equivalent:

```
  reco->Draw("slc.stp[3]","","",1,0);
```

This command seems like it should draw the `*stp` array index for the 3rd CandStrip in each CandSlice (looping over CandSlice), but in actuality it loops over all `*stp` elements of the 3rd CandSlice object. ROOT apparently does not care where the square brackets are located, instead assigning the square brackets by order found. In this case the first square bracket ([3]) corresponds to the first array (`slc`), making the two expressions above equivalent.

If one wanted to draw the `*stp` array index for the 3rd CandStrip in each CandSlice, one would do

```
  reco->Draw("slc[].stp[3]","","",1,0);
```

The presence of the first pair of square brackets ensures that the [3] corresponds to the second array in this expression (in this case, `slc.stp`). Alternatively, one can do

```
  reco->Draw("slc.stp[][3]","","",1,0);
```

One can now use this index to plot information associated with StripSRTTree. For example, to plot the transverse versus longitudinal positions for all CandStrip objects in the first CandSlice:

```
  reco->Draw("stp[slc[0].stp].tpos:stp[slc[0].stp].z");
```

More examples of the use of `stp` indices are given in the next section.

One thing to note about CandSlice objects is that while it is true that all CandStrip objects are found in a CandSlice (at least for the far detector), it is not true that all CandDigits are found in a CandStrip. Whether a CandDigit is to be included in a CandStrip depends on the algorithm. One possible reason why a CandDigit may not be included a CandStrip (and therefore a CandSlice) is because it is flagged as a crosstalk digitization. The user is referred to the specific algorithms and packages if interested.

2.7 TrackSRTTree

The TClonesArray `*trk` contains elements of type TrackSRTTree. Each element of `*trk` corresponds to a single 3D track. The structure of TrackSRTTree shown below:

```
class TrackSRTTree : public TObject
{
  ...
  UShort_t index;
  Int_t ndigit;
  Int_t nstrip;
  Int_t *stp; //[nstrip]
  Float_t *stpu; //[nstrip]
  Float_t *stpv; //[nstrip]
  Float_t *stpx; //[nstrip]
  Float_t *stpy; //[nstrip]
  Float_t *stpz; //[nstrip]
  Float_t *stpt0; //[nstrip]
  Float_t *stpt1; //[nstrip]
  Float_t *stpph0sigmap; //[nstrip]
  Float_t *stpph0mip; //[nstrip]
  Float_t *stpph0gev; //[nstrip]
  Float_t *stpph1sigmap; //[nstrip]
  Float_t *stpph1mip; //[nstrip]
  Float_t *stpph1gev; //[nstrip]
  Float_t *stpds; //[nstrip]
  Float_t *stpattn0c0; //[nstrip]
  Float_t *stpattn1c0; //[nstrip]
  Float_t *stptcal0t0; //[nstrip]
  Float_t *stptcal1t0; //[nstrip]
  Bool_t *stpfit; //[nstrip]
  Bool_t *stpfitchi2; //[nstrip]
  Bool_t *stpfitprechi2; //[nstrip]
  Bool_t *stpfitqp; //[nstrip]
  DigitStripPulseHeightSR ph;
  TrackPlaneInfoSR plane;
  VertexInfoSR vtx;
  VertexInfoSR end;
  VertexInfoSR lin; // linear fit
  FiducialInfoSR fidvtx;
  FiducialInfoSR fidend;
  FiducialInfoSR fidall;
  TrackTimeInfoSR time;
  Float_t ds;
  Float_t range;
  Float_t cputime;
  MomentumInfoSR momentum;
  FitTrackInfoSR fit;
};
```

Note the presence of the CandStrip array `*stp`. Each TrackSRTTree object has multiple arrays, all with the same index. Each array that begins with `stp` indicates that it is an array over CandStrip objects or information associated with strips.

As an example of using the `*stp` index, we examine a multiple muon event in the far detector. Figure 2 shows the transverse versus longitudinal positions of the hit strips. The two tracks (one in red, one in blue) are clearly visible. The hollow circles indicate CandStrip objects that were not included as part of any track. The commands used to generate this plot are as follows:

```
[1] reco->SetMarkerStyle(24);
[2] reco->Draw("stp.tpos:stp.z","","",1,0);
[3] reco->SetMarkerStyle(20);
```

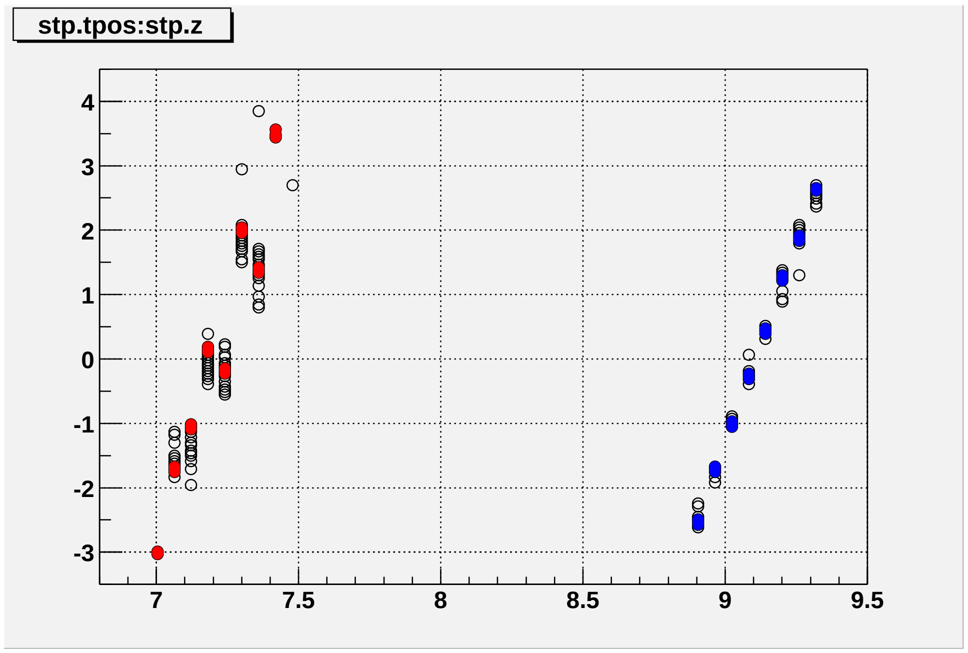

Figure 2: tpos vs z for CandTrack objects

```
[4] reco->SetMarkerColor(2);
[5] reco->Draw("stp[trk[0].stp].tpos:stp[trk[0].stp].z","","same",1,0);
[6] reco->SetMarkerColor(4);
[7] reco->Draw("stp[trk[1].stp].tpos:stp[trk[1].stp].z","","same",1,0);
```

Line [2] draws the strip positions for all CandStrip objects. Lines [5] and [7] draw the positions only for those CandStrip objects found in tracks. If we had wanted to draw the strip positions for strips in any tracks, we could have done

```
reco->Draw("stp[trk.stp].tpos:stp[trk.stp].z","","",1,0);
```

In addition to accessing StripSRTTree information through the `*stp` index, there is strip associated information which is specific to each CandTrack object. For example, given a 3D track, we can then calculate the 3D position at each hit plane (`*stpu`, `*stpv`, `*stpx`, `*stpy`).

For clarity, we use as an example the first track (in red) shown in Fig. 2.

```
[1] reco->Scan("trk[0].stp[9]","","",1,0)
************************
```

```
*    Row   * trk[0].st *
************************
*        0 *        77 *
************************
(Int_t)1
[2] reco->Scan("stp[77].tpos:stp[77].planeview","","",1,0)
************************************
*    Row   * stp[77].t * stp[77].p *
************************************
*        0 * 1.3804974 *         2 *
************************************
(Int_t)1
[3] reco->Scan("trk[0].stpu[9]:stp[trk[0].stp[9]].tpos","","",1,0)
************************************
*    Row   * trk[0].st * stp[trk[0 *
************************************
*        0 * 1.3812383 * 1.3804974 *
************************************
```

Line [1] shows that the CandStrip which corresponds to element 9 in trk[0] corresponds to element 77 in the TClonesArray *stp. From line [2] we see that the transverse position is 1.38 m and that it is a u-view strip (planeview == 2). Thus we can compare trk[0].stpu[9] with stp[77].tpos and we see that they are about the same (they are not exactly equal because 3D positions found by tracks take a pulse height weighted average of strips in a plane). More illustrative is that instead of referring to stp[77].tpos, we can use instead stp[trk[0].stp[9]].tpos"; this makes it simple to combine the additional strip information found in TrackSRTTree with strip information found in StripSRTTree by using the same index.

It was found that older versions of ROOT may not correctly handle indices and multiple expressions (such as in line [3] above). In particular, if one replaces the Scan with a Draw method in line [3], it may be seen that the resultant plot is incorrect. This has been fixed in newer versions of ROOT (beginning with 3.03/09).

There are several strip associated variables in TrackSRTTree. The previous discussion used *stpu as an example. A listing and description of all available variables is found in Table 1.

There are several other objects in TrackSRTTree. The variable plane is of class TrackPlaneInfoSR and contains information about the number of hit planes in the track and the begin and end planes, both summed over all plane views and separately. Note that the begin plane here is not necessarily the upstream plane, as timing information is used to determine the start of the track. The variable ntrklike is the number of hit planes in the track which are relatively clean (free from the presence of additional hits).

Name	Description
stp	index over TClonesArray `*stp`
stpu, stpv,	u, v, x, y, z positions at hit plane
stpx, stpy, stpz	
stpt0, stpt1	3D times for east and west ends, corrected for
	propagation delays
stpph0sigmap,	strip pulse height for east and west ends, corrected
stpph1sigmap	for fiber attenuation
stpph0mip, tpph1mip	strip pulse height for east and west ends, in MIP units
stpph0gev, stpph1gev	strip pulse height for east and west ends, in GeV units
stpds	track path length from track end
stpattn0c0, stpattn1c0	C0 for east and west ends from mapper results
stptcal0t0, stptcal1t0	T0 offsets for east and west ends
stpfit	1 if this hit strip was used in final track fit
stpfitchi2	final χ^2 at this strip
stpfitprechi2	pre χ^2 at this strip from Kalman filter
stpfitqp	charge over momentum at this strip from fit

Table 1: TrackSRTTree `*stp` variable descriptions

There are three variables of class VertexInfoSR: `vtx`, `end`, and `lin`. The first and second variables contain information at the begin and end track positions, while the last holds information from a linear fit to the track hit positions. The structure of VertexInfoSR is

```
class VertexInfoSR : public TObject
{
  ...
  Float_t u;
  Float_t v;
  Float_t x;
  Float_t y;
  Float_t z;
  Float_t t;
  Int_t plane;
  Float_t eu;
  Float_t ev;
  Float_t ex;
  Float_t ey;
  Float_t dcosu;
  Float_t dcosv;
  Float_t dcosx;
  Float_t dcosy;
  Float_t dcosz;
  Float_t edcosu;
  Float_t edcosv;
  Float_t edcosx;
  Float_t edcosy;
  Float_t edcosz;
};
```

VertexInfoSR variables which begin with the letter 'e' are the uncertainties from the track fit.

The class FiducialInfoSR contains information about fiducial containment:

```
class FiducialInfoSR : public TObject
{
  ...
  Float_t dr;
  Float_t dz;
  Float_t trace;
  Float_t tracez;
  Int_t nplane; // number of planes extrapolated to beg/end hit planes
  Int_t nplaneu; // number of planes extrapolated to beg/end hit planes
  Int_t nplanev; // number of planes extrapolated to beg/end hit planes
};


The member variables {\tt dr} is the minimum transverse distance to the
detector edge (a positive value indicates containment within the detector);
{\tt dz} is the minimum longitudinal distance to the detector edge (note
that the validity of this variable depends on Plex and UgliGeometry knowing
where the active detector ends).

The two variables {\tt trace} and {\tt tracez} represent the path length
(in meters) between the track vertex or end and the detector edge.  If
a magnetic track fitting package has been included in the reconstruction,
these represent the swum values in the magnetic field.

There are some cases in which the tracker does not pick up all hits along
the track, for example when some planes are not properly demuxed.  By
analyzing the longitudinal energy distribution one may make a guess as to
where the track end points actually are.  Some tracking packages will
attempt to swim or project the track from its two points where it stopped
tracking to the track ends based on longitudinal energy distributions.
The number of planes over which a projection is performed is stored
in {\tt nplane}, and in {\tt nplaneu} and {\tt nplanev} for the $u$ and
```

```
$v$ views separately.



There are
3 member objects of TrackSRTTree of class FiducialInfoSR: {\tt fidvtx},
{\tt fidend}, and {\tt fidall}.  The first two represent containment
information at the vertex and end positions of the track.  The last,
{\tt fidall}, contains information about the distance of closest approach
over all 3D points on the track, with the transverse and longitudinal
information calculated independently.  So, for example, given a track
which enters the far detector through the first plane at the coil hole,
bends out and grazes the transverse edge of the detector, and then bends
back in and stops near the coil hole, {\tt fidall.dz} and {\tt fidall.dr} would
both be 0, while amongst {\tt fidvtx.dz}, {\tt fidvtx.dr}, {\tt fidend.dz}, and
{\tt fidend.dr} only the first would be 0.

The member variable {\tt time} of class TrackTimeInfoSR holds timing
information for a particular track:

\begin{verbatim}
class TrackTimeInfoSR : public TObject
{
  ...
  UShort_t ndigit;
  Float_t chi2;
  Float_t u0;
  Float_t u1;
  Float_t v0;
  Float_t v1;
  Float_t cdtds; // 1/beta
  Float_t dtds;
  Float_t t0;
  Float_t du; // difference between timing based position and spatial position
  Float_t dv; // difference between timing based position and spatial position
};
```

The variable `ndigit` represents the number of digitizations used in determining the track direction from timing. Not all digitizations are necessarily used, as some may digit times may be outliers, or some digits may not represent the earliest time of a phototube (only the time of the first signal above threshold of a multianode phototube is recorded, at least for the far detector). The variable `chi2` represents the χ^2 of a fit done in the track direction determination. The four variables `u0`, `u1`, `v0`, and `v1` hold the mean values of the fully calibrated and propagation delay corrected times for the separate plane views and separate strip ends (in the case of double ended read out). The velocity of the track is measured by comparing the corrected times to the travel distance; the absolute value of the inverse of the track velocity (normalized to the speed of light) is then calculated and stored in `cdtds`. From the time fit which determines $1/\beta$, we have the unnormalized slope `dtds` and the offset `t0`.

For planes in the far detector which have digits at both ends, the spatial position along the scintillator strip is calculated based on the timing difference; this is compared to the actual position based on strip locations in the opposite view, the difference (in meters) is then stored in `du` and `dv`.

The total path length of the track in the detector is given by `ds`, and the amount of material that the track traversed is given by `range` (in g/cm^2). The momentum of the track from range is calculated and stored in the `range` variable in `momentum` of class MomentumInfoSR. Note that this value for the momentum from range is calculated in the same way regardless of whether or not a track stops in the detector. If magnetic track fitting has been done, the momentum from curvature in the magnetic field as well as the uncertainty from the fit are available through the variables `momentum.qp` and `momentum.eqp`.

The total amount of time spent in the tracking package which produced the tracks in this tree is calculated and stored in `cputime`.

Finally, additional information from the track fitting is stored in the class FitTrackInfoSR:

```
class FitTrackInfoSR : public TObject
{
  ...
  Bool_t pass;
  Float_t chi2;
  Int_t ndof;
  Int_t niterate;
  Int_t nswimfail;
  Float_t cputime; // execution time in seconds
}

Note that some of these variables are currently defined only for the
CandFitTrackSR package.


\subsection{ShowerSRTTree}

Reconstructed shower information is contained in the class
ShowerSRTTree, whose structure is
```

```
\begin{verbatim}
class ShowerSRTTree : public TObject
{
  ...
  UShort_t index;
  Int_t ndigit;
  Int_t nstrip;
  Int_t *stp; //[nstrip]
  DigitStripPulseHeightSR ph;
  PlaneInfoSR plane;
  VertexInfoSR vtx;
};
```

Currently this class is not as developed as TrackSRTTree. All of the member variables of ShowerSRTTree are also found in TrackSRTTree, and will therefore not be described here.

2.8 EventSRTTree

Reconstructed showers and tracks are associated together based on proximity in space as well as time to form candidate events. In general a reconstructed candidate event may contain any number of tracks and showers (by construction an event will contain at least one track or shower). Much of the information that users would want about reconstructed events are actually contained in the tracks and showers that make up the events, and so perhaps the most useful information contained in EventSRTTree are indices over reconstructed tracks and showers.

The TObjArray `*evt` is an array of reconstructed events of class EventSRTTree:

```
class EventSRTTree : public TObject
{
  ...
  UShort_t index;
  Int_t ndigit;
  Int_t nstrip;
  Int_t *stp; //[nstrip]
  Int_t ntrack;
  Int_t *trk; //[ntrack];
  Int_t nshower;
  Int_t *shw; //[nshower];
  DigitStripPulseHeightSR ph;
  PlaneInfoSR plane;
  VertexInfoSR vtx;
  VertexInfoSR end;
};
```

The index *stp is an index over the StripSRTTree array *stp in the top level class TTreeSR. Note that an event may contain strips which are not in any track or shower constituting the event. Also, the index evt[i].stp[] does not double count strips, whereas it is possible for either index evt[i].trk[].stp[] or evt[i].shw[].stp[] to count the same strip twice (if, for example, an event contains two tracks which share the same strip).

The indices *trk and *shw work in the same way as the index *stp. For example, to draw the momentum from range for all tracks which belong to the first reconstructed event for snarl i:

```
  reco->Draw("trk[evt[0].trk].momentum.range","evthdr.snarl==i");
```

2.9 MCTruthSR

Summary truth information for monte carlo events are stored in the MCTruthSR class which has the branch name mc:

```
class MCTruthSR : public TObject
{
  ...
  Float_t vtxx;
  Float_t vtxy;
  Float_t vtxz;

  Int_t inu;
  Int_t inunoosc;
  Int_t itg;
  Int_t iboson;
  Int_t iresonance;
  Int_t iaction;
  Float_t a;
  Float_t z;
  Float_t sigma;
  Float_t p4neu[4];
  Float_t p4neunoosc[4];
  Float_t p4tgt[4];
  Float_t p4shw[4];
  Float_t p4mu1[4];
  Float_t p4mu2[4];
  Float_t p4el1[4];
  Float_t p4el2[4];
  Float_t p4tau[4];
  Float_t x;
  Float_t y;
  Float_t q2;
  Float_t w2;
  Float_t emfrac;
}
```

Currently these are filled in only for the first NeuVtx and NeuKin objects (from the REROOT_Classes package). These variables are taken straight from those classes, so interested parties can find the definition in these two classes.

2.10 FLSDigitSR

Digit truth information for monte carlo events can be found in the `flsdgt` branch which is of the class FLSDigitSR:

```
class FLSDigitSR : public TObject
{
  ...
  Int_t      Plane;
  Int_t      Strip;
  Float_t    TPos;
  Float_t    RawA;
  Float_t    RawB;
  Float_t    CorrA;
  Float_t    CorrB;
  Float_t    CorrSum;
  Float_t    TDCA;
  Float_t    TDCB;
  Int_t      TubePixelA;
  Int_t      TubePixelB;
  Float_t    SignalPEA;
  Float_t    SignalPEB;
  Float_t    InitialTDCA;
  Float_t    InitialTDCB;
  Float_t    SumETrue;
  Float_t    AveDistTrueA;
  Float_t    AveDistTrueB;
  Int_t      HitBits;
}
```

These variables are lifted directly from the FLSDigit class in REROOT_Classes.

Note that the default behavior is to not write out this branch. Users who wish to have this branch written out should set the parameter WriteFLSDigit to 1 in EventSRListModule.

References

[1] NuMI-NOTE-COMP-916 A Description of the MINOS Reconstruction Framework