Data Storage in g2MIGTRACE Root Files

Kevin Lynch

2009-09-17

1 Introduction

In this short note, I briefly document the g2MIGTRACE simulation data that may be available in the Root output files.

By default, each "run" of the simulation (generally, and invocation of /run/beamOn), results in a file with a name of the form g2MIGTRACE_<date string>_run<number>.root. The date string is the GMT date/time of the start of the run in ISO8601 format (without separators), while the number is a zero based count of the run within the current invocation of the executable. The basename is modifiable, so the initial filename string may not match the simulation name.

In any given file, there are a number of guaranteed and optional trees, whose presence depends on various flags set at simulation run time. Each primary event has a corresponding (potentially empty) entry in every tree.

2 General Structure for Reading

So, how do you read these trees and do something useful with them? You need to get the code for the simulation, and build the libROOTRecords shared library. This library contains the dictionary that enables Root to stream the storage classes into and out of the TFile, as well as handle command line manipulation in the Root interpreter.

When reading the data, I find it generally easier to avoid the command line, and build a binary that links against this shared library. Explaining the intricacies here is beyond the scope of this note (but see the appendix for an unexplained example); see your local Root/C++ Guru for help.

Your code will need to:

1. Open and check the validity of the Root file,
2. Locate and connect to the independent (that is, non-TTree) date in that file,
3. Locate and connect to the TTrees in that file,
4. Allocate the in memory representations of the data types you need,
5. Connect those representations to the TTrees via TTree::SetBranchAddress,
6. Iterate over the data, forming whatever output you wish, and finally
7. Cleaning up and closing down.

There will eventually be examples of doing this in both the distribution, and in this manual. But they aren't here now.

3 Non-Tree Data

In addition to the trees, there are run metadata elements that are stored in the TFile, but not in TTrees.

3.1 Object Manager

The class `g2UniqueObjectManager` stores a persistent mapping between the physical objects in a run (such as *Calorimeter[01]* or *VacuumChamberWalls[10]* and an integral Unique Identifier that is used in the various trees to identify volumes where events occurred. A snapshot of this mapping is made at the *BeginOfRun* transition, where the geometry is frozen. The stored manager can then be used to determine the volume identity of an event by lookup of the UID, or regular expression matching. See the example in the appendix.

```
class g2UniqueObjectManager : public TObject {
public:
  bool add(void*, std::string);
  void clear();
  std::string lookup(ULong64_t) const;
  bool re_match(ULong64_t, std::string) const;
  std::vector<std::string> re_match_names(std::string) const;
  std::vector<ULong64_t> re_match_uids(std::string) const;
  int count() const;
};
```

4 Guaranteed Trees

There are two trees that are currently guaranteed to exist in any output file: an Event Record, storing event metadata, and a Track Map, storing the relationships between all Geant track identifiers and their birth conditions.

4.1 Event Record

The `TTree` *eventStatusTree* contains the event status metadata. The tree entry contains a single `eventRecord` structure, with the following public interface:

```
struct eventRecord {
  Bool_t muWasStored;
  Int_t lastTurn;
};
```

The boolean is used during injection studies, to determine if the injected muon was considered stored according to the settings chosen during the simulation run. The integer records the last value of the turn counter when the event was terminated, whether stored or not; this is useful, for example, for preliminary loss studies.

4.2 Track Map

The `TTree` *trackTree* contains a `std::map`, *trackIDMap* between the integer *trackID* of each particle track and a `trackRecord` object:

```
struct trackRecord {
  std::string trackType;
  Int_t trackID, parentTrackID, turn, volumeUID;
  Float_t rhat, vhat, theta, time;
  Float_t p, prhat, pvhat;
};
```

As many of the variables listed have a common usage in other data structures, some detailed discussion is in order here:

- The *trackType* stores a textual representation of the particle type, as known to Geant (*mu+*, *e-*, etc.); we could have equivalently used an integral ID such as the PDG particle code, but the space savings was deemed not worth the extra effort.

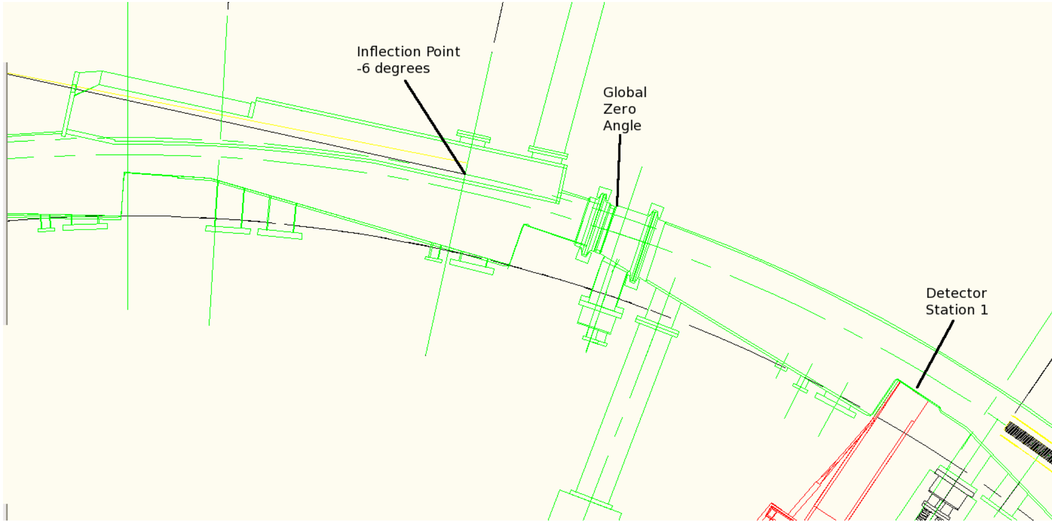

Figure 1: The definition of the global zero angle.

- The *trackID* stores the unique Geant ID of the particle track; this equals 1 for the primary. The *parentTrackID* is the ID for the particle that caused the creation (through a Geant Physics Process) of the current particle.
- The *turn* is the current turn after injection of this particle's birth. That is, if the particle is a decay electron, born while the muon was on it's fourth circuit of the ring, turn == 3.
- The *volumeUID* is the unique identifier of the physical volume of the particle's birth. This UID can be used to look up the physical volume name in the `g2UniqueObjectManager` stored in the Root file.
- The remaining six variables are the global coordinates and momenta of the particle's birth:
  - rhat is the radial distance from the ideal storage orbit,
  - vhat is the vertical distance from the ideal storage orbit,
  - theta is the angular distance downstream of the global zero angle (see below), and
  - time is the global time.
  - p is the total momentum,
  - phat (pvhat) is the radial (vertical) fraction of the momentum.

The global coordinate system is defined for convenience in constructing the simulation. The storage orbit is at a radius of 7.112 m, zero vertical offset, with the zero angle defined at the upstream end of the first bellows downstream of the inflection point; see Figure 1.

5 Optional Trees

There are a number of trees that may be enabled during a run of the simulation, and some of those may be further restricted at run time in the amount of information they collect. These optional trees are described here.

5.1 Inflector Tracking Data

The `TTree` *inflectorTree* contains the `TBranch` *inflectorVector*, which is a `std::vector<inflectorRecord>` of inflector tracking volume hits. A run time selectable number of tracking planes, labelled *InflectorTracker[nn]* are inserted into the inflector channel volume, with *nn* a two digit value increasing from 0 at the upstream end. Hits on these massless tracking planes result in an entry of type summarized here:

```
struct inflectorRecord {
  Float_t x_inf, y_inf, z_inf;
  Float_t px_inf, py_inf, pz_inf;
  Float_t time; // ns from THIS particle's parent injection
  Int_t trackID, volumeUID;
};
```

5.2 Ring Tracking Data

The `TTree` *trackerTree* (not to be confused with the *trackTree* above!) contains the single `TBranch`, *trackerVector*, a `std::vector<trackerRecord>` of tracking volume hits in the storage volume. There are a dozen tracking volumes, spaced evenly around the ring, which cover the entire cross section of the interior of the vacuum region. Hits on these massless tracking planes result in an entry of type summarized here:

```
struct trackerRecord {
  Float_t rhat, vhat, theta, time;
  Float_t p, prhat, pvhat;
  Int_t turn, volumeUID, trackID;
};
```

5.3 Ring Energy Loss Data

The `TTree` *ringHitTree* contains a single `TBranch` *ringHitVector*, a `std::vector<ringRecord>` of energy loss events. These can come from any physical object in the vacuum vessel (including plates, inflector, supports, etc.). There are a few hundred of these physical objects. Any hit resulting in non-zero energy loss results in an entry of the type summarized here:

```
struct ringRecord {
  Float_t rhat, vhat, theta, time;
  Float_t p, prhat, pvhat;
  Int_t turn, volumeUID, trackID;
  Float_t deltaP, deltaPrhat, deltaPvhat, deltaE;
};
```

The momentum deltas are absolute, not fractional.

It is with these hits that the regular expression matching built into the `g2UniqueObjectManager` comes in handy, allowing matching against name patterns ("all quadrupoles", "only the first kicker", etc). See the example in the Appendix.

5.4 Calorimeter Data

The `TTree` *caloHitTree* contains a single `TBranch` *caloHitVector*, a `std::vector<caloRecord>` of calorimeter hits from any particle type. The particles are currently "swallowed" by the calorimeter, resulting in the record of the complete momentum, position, and global time. The momentum and position are stored in the local coordinates of the calorimeter, not the global coordinates of the storage ring!

```
struct caloRecord {
  Int_t turn, caloNum, trackID;
  Float_t r, t, v, time;
  Float_t pr, pt, pv;
};
```

The coordinate system is defined as follows:

- r is the "most radial like" direction, increasing in the global, radially outward direction,

- t is the “thickness” direction, normal to both the other vectors, and increasing in the “most downstream like” direction, an
- v is the “most vertical” direction, increasing in the globally upward direction.

The coordinate $(0, 0, 0)$ is at the geometric center of the calorimeter block. Then, for a calorimeter of size $21\,\text{cm} \times 12\,\text{cm} \times 15\,\text{cm}$, the center of the upstream face will be located at $(0, -6\,\text{cm}, 0)$.

6 Other Data

There is currently no other data available, but may be in future.

A Example Analysis

Listing 1: An example analysis of a `TTree` from g2MIGTRACE

```
1  #include <string>
2  #include <iostream>
3
4  #include "TFile.h"
5  #include "TTree.h"
6  #include "TH1.h"
7  #include "TCanvas.h"
8  #include "TSystem.h"
9
10 #include <string>
11 #include <sstream>
12 #include <iomanip>
13
14 #include "array_macros.hh"
15
16 #include "ringRecord.rhh"
17 #include "g2UniqueObjectManager.rhh"
18
19
20 struct info {
21   std::string name;
22   std::string title;
23   std::string match;
24   double max;
25   TH1 *hist;
26   int hits;
27 };
28
29 info infos[] = {
30   {"dswindow", "Downstream Window", "DownstreamWindow", 5., 0, 0},
31   {"dsflange", "Downstream Flange", "DownstreamEndFlange", 5., 0, 0},
32   {"dsal", "Downstream Aluminum", "DownstreamEquivalentAl", 5., 0, 0},
33   {"dscu", "Downstream Copper", "DownstreamEquivalentCu", 5., 0, 0},
34   {"dsnbti", "Downstream NbTi", "DownstreamEquivalentNbTi", 5., 0, 0},
35   {"dsall", "Downstream All", "Downstream", 5., 0, 0},
36   //
37   {"uswindow", "Upstream Window", "UpstreamWindow", 5., 0, 0},
38   {"usflange", "Upstream Flange", "UpstreamEndFlange", 5., 0, 0},
```

```
39    {"usal", "Upstream Aluminum", "UpstreamEquivalentAl", 5., 0, 0},
40    {"uscu", "Upstream Copper", "UpstreamEquivalentCu", 5., 0, 0},
41    {"usnbti", "Upstream NbTi", "UpstreamEquivalentNbTi", 5., 0, 0},
42    {"usall", "Upstream All", "Upstream", 5., 0, 0},
43    //
44    {"cryowindow", "Cryostat Window", "PerpendicularCryostatWall", 5., 0, 0},
45    //
46    {"quadplate", "First Outer Quadrupole Plate",
47     "QuadOuterPlate\\[0\\]\\[0\\]", 15., 0, 0},
48    {"quadsupports", "First Outer Quadrupole Plate Supports",
49     "QuadOuterSupport\\[0\\]\\[0\\]", 15., 0, 0},
50    {"quadall", "First Outer Quadrupole All",
51     "QuadOuterPlate\\[0\\]\\[0\\]|QuadOuterSupport\\[0\\]\\[0\\]", 15., 0, 0},
52    {"all", "All Injection Losses", "", 30., 0, 0},
53  };
54
55  void energy_loss(std::string fname){
56
57    TFile f(fname.c_str(), "READ");
58    if( f.IsZombie() ){
59      std::cout << "File " << fname << " doesn't exist.  That's a problem!\n";
60      return;
61    }
62
63    g2UniqueObjectManager *uom =
64      static_cast<g2UniqueObjectManager*>( f.Get("uom") );
65
66    TTree *eventStatus =
67      static_cast<TTree *>( f.Get("eventStatusTree") );
68    if( !eventStatus ){
69      std::cout << "eventStatusTree missing in file " << fname << ".  Oops.\n";
70      return;
71    }
72
73    TTree *ringHit = static_cast<TTree *> (f.Get("ringHitTree"));
74    if( !ringHit ){
75      std::cout << "ringHitTree missing in file " << fname << ".  Oops.\n";
76      return;
77    }
78
79    std::vector<ringRecord> const *rrv = new std::vector<ringRecord>;
80    ringHit->SetBranchAddress("ringHitVector", &rrv);
81
82
83    std::ostringstream o;
84    // init the canvases and histos
85    TCanvas *canvas = new TCanvas("canvas", "canvas", 0, 0, 500, 400);
86    for(int i=0; i!=dimensionof(infos); ++i){
87      infos[i].hist = new TH1D(infos[i].name.c_str(), infos[i].title.c_str(),
88                               100, 0., infos[i].max);
89    }
90
91    // loop and fill
92    //  for( int i = 0; i!=100; ++i ){
```

```
93   for( int i = 0; i!=ringHit->GetEntries(); ++i ){
94     if( i%1000 == 0 )
95       std::cout << "Processing Entry " << i << " of " << ringHit->GetEntries() << '\n';
96     ringHit->GetEntry(i);
97     int dinfos = dimensionof(infos);
98     double accum[dinfos];
99     for(int j=0; j!=dinfos;++j)
100      accum[j] = 0.;
101    std::vector<ringRecord>::const_iterator b = rrv->begin(), e = rrv->end();
102    while( b!= e ){
103      // don't regex the last one!
104      for(int j=0; j!=dinfos-1; ++j){
105        if( uom->re_match(b->volumeUID, infos[j].match) ){
106          accum[j]-= b->deltaE;
107          accum[dinfos-1] -= b->deltaE;
108          ++(infos[j].hits);
109          ++(infos[dinfos-1].hits);
110        }
111      }
112      ++b;
113    }
114    for(int j=0; j!=dinfos; ++j){
115      if(accum[j] != 0.)
116        infos[j].hist->Fill(accum[j]);
117    }
118  }
119
120
121  for(int i=0; i!=dimensionof(infos); ++i){
122    std::cout << "Hits for " << infos[i].title << ' ' << infos[i].hits << '\n';
123    canvas->Clear();
124    infos[i].hist->Draw();
125    o.str("");
126    o << infos[i].name << ".eps";
127    canvas->SaveAs(o.str().c_str(), "");
128    o.str("");
129    o << infos[i].name << ".png";
130    canvas->SaveAs(o.str().c_str(), "");
131    o.str("");
132    o << infos[i].name << ".C";
133    infos[i].hist->SaveAs(o.str().c_str(), "");
134    delete infos[i].hist;
135  }
136  delete canvas;
137
138 }
```