Information on generators for the KamLAND simulation

G.Horton-Smith

October 29, 2001

1 Overview

This note contains a description of the primary event generator model and the event generator components for the KamLAND simulation (KLG4sim).

A primary event is a list of particles and their initial momenta and their starting points in space and time. It is convenient to split a primary event into three parts: a global start time; a global position; and a "vertex" which specifies the momenta and, when necessary, a relative spatial offset and relative time offset for each particle.

A listing of all expected event types shows that we need 2 styles of global start time generation (Table 1), 12 global position generators (Table 2), and 19 vertex generators (Table 3) to support 51 different sources of primary events currently envisioned (Table 4).

In KLG4sim, the task of global position generation is handled by "global position generators" (implemented as objects which have `KLVPosGen` as a base), and the task of vertex generation is handled by "vertex generators" (implemented as objects which have `KLVVertexGen` as a base). There are two principal reasons for this: (1) it allows a relatively small number of position and vertex generators to be reused; (2) it allows the physics to be cleanly separated from the geometry.

Primary event generation in KLG4sim is performed by first choosing an event type and global time, then calling a vertex generator, then applying the global position generator. The primary track information from the vertex generator is available to the position generator, which is needed for example when generating random impact parameters to external cosmic rays.

All of these generators are available to KLG4sim at the same time, and the user may specify the rate for each event type individually. The event times are chosen and events mixed appropriately, including such effects as accidental coincidences and accidental disruption of real coincidences by other events. This is the task of `KLPrimaryGeneratorAction`.[1] Of course a user is free to zero the rates of all but one event type in order to get a simpler Monte Carlo.

Most vertex generators are currently envisioned as external applications feeding events to the simulation via an ASCII stream. For vertices which have no relative time or position offsets of the initial particles, the Geant4 `G4HEPEvtInterface` mechanism would have been sufficient. In order to allow for the case where time and/or position offsets between primary particles are required, and to support the communication of special information from generator to the KLG4sim output stream, an enhanced vertex genera-

[1] Suspension of highly delayed particles to later events still not quite finished as of 2001-08-31.

tor interface (KLVertexGen_HEPEvt) has been written which supports a superset of the G4HEPEvtInterface format. This is described in more detail below.

The Palo Verde Fortran library of radioactivity generators can be used with the KLVertexGen_HEPEvt mechanism. Geant4 also has a complete radioactive decay process for any isotope, which may offer improved flexibility and user control. See http://www.-space.dera.gov.uk/space_env/rdm.html. It is currently envisioned to use the Fortran generators initially, and to migrate to the native Geant4 generators if and when their accuracy and utility can be demonstrated.

Note that calibration source event generators are not included on the list. A spare KLVertexGen_HEPEvt interface is provided which can be used for supplying calibration source events to the simulation without rewriting KLG4sim.

If you have written or are writing a generator for a primary event type not listed in Table 4, please let me know so that it can be added to this list.

2 Description and user controls for the master event generator (including global time generator)

The event time sequence generation is handled in the master primary event generator, KLPrimaryGeneratorAction, which then calls the appropriate vertex and position generators to fill in the important details of each primary event. The time sequence is determined by the user-specified event rates, the global event time window, and the time generator code set for each event type, also referred to as the "trigger flag" for that event.

If two or more simple events happen within the time window (200 ns by default), they will be combined into a single event before being sent to the Geant4 kernel. This is so that photoelectrons with overlapping time distributions are not artificially sorted into events according to what vertex they came from. Eventually, any tracks in an event with time offsets outside the event time frame (*e.g.*, from a delayed radioactive decay) will be removed to form a separate event. This is to avoid having to simulate an entire day of data as one Geant4 event.

Each event type also has a trigger flag which specifies whether an event type can be seen independently or only as "pile-up" on another event. Examples of the latter might include wall activity in the OD or C-14 in the scintillator.

One way of summarizing the master primary event generator is to list the information it keeps in memory:

- the current "universal time";
- the user-specified event time window;
- the rate for each event type (indexed by the code in Table 4);
- the trigger flag for each event type (see above);
- a list of vertex generators and their English-language names (indexed by the code in Table 3);
- a list of position generators and their English-language names (indexed by the code in Table 2);
- a list of which vertex and position generator to use for each event type (Table 4 itself).

Below are short descriptions of the commands that can be used to control the master event generator. The on-line help is actually more extensive for some commands.

/generator/list This will list the names and codes for all generators in your copy of KLG4sim, basically producing an as-compiled version of Table 4.

/generator/rates ***[index rate-Hz [flag]]*** With no options, lists the indices, rates, and names for each event type. If an event type has the "pile-up only" flag set, the rate is marked with an asterisk ('*'). To change the rate of an event type, specify the index and rate in Hz, optionally followed by a 1 or 0 to set or clear the "pile-up only" trigger flag.

/generator/gun ***particle-name x y z px py pz K [polx poly polz]*** This command is a convenient way of setting the user-controlled test gun parameters. It is really an alias for the two commands:

```
/generator/pos/set 9 x y z
/generator/vtx/set 17 particle_name px py pz K polx poly polz
```

For isotropic direction vectors, leave *(px,py,pz)* all zero and set K. For fancier options, such as uniformly filling a volume or uniformly painting a surface, use `/generator/pos/set 9` directly. (See the section on position generators below.)

3 Description and user controls for position generators

There are actually only three styles of position generator currently implemented: `KLPosGen_null`, `KLPosGen_Cosmic`, and `KLPosGen_PointPaintFill`. All of the specific position distributions listed in Table 2 can be generated by an appropriate setting of the state of one of these position generators. Every position generator provides an interface for setting and retrieving its state as a text string.

/generator/pos/set ***index [setting]*** is the fundamental command for controlling the state of the position generators. The *index* parameter is the index code from Table 2.[2] The *setting* parameter is the text string representation of the generator state, which generally must be enclosed in quotation marks. If the *setting* parameter is omitted, the command prints the current setting along with documentation on what the state string means for that generator.

Below are descriptions of the three styles of position generators.

KLPosGen_null does nothing, and is useful only if the vertex generator sets the initial positions of the tracks. In this case, the vertex positions are interpreted as absolute coordinates rather than relative to a relocatable primary event location.

KLPosGen_Cosmic generates positions uniformly distributed over an area normal to the incident direction of first track in the vertex; these are then offset to the corresponding entrance point of the track into the world volume. The format of the state string is "*width_mm height_mm*", giving the width and height (in mm) of a rectangular area normal to the primary track's incident direction. Appropriate values for KamLAND would be 20000 33000. The rectangle is rotated so that the "width" direction vector lies in the XY plane for non-zero polar angle of the incident track.

[2] This is also the code in the "position" column in Table 4 and the code in the `pos` column of the output from the `/generator/list` command described previously.

If a track is generated which completely misses the detector, the vertex tracks are retained but modified such that Geant4 does not attempt to track them. This means the generated external flux I in Hz/mm^2 is always given by $I = f/(wh)$, where f is the rate set via `/generators/rate`, w is the width, and h is the height, regardless of the geometry of the detector. Rate must be chosen appropriately for the area of the rectangle.

KLPosGen_PointPaintFill is a multi-purpose "Swiss-army knife" position generator which can be used to generate positions at (1) a fixed point, (2) positions uniformly filling a physical volume, (3) positions uniformly filling any occurance of a given material within a physical volume or its daughters, or (4) positions uniformly "painting" the surface of a physical volume, optionally with non-zero thickness of the paint, optionally restricted to points occupied by a given material. All of this is done using the same Geant4 geometry data and routines as are used by the tracking. The format of the state string is

(1) "*x y z*", the position of the point in mm.
(2) "*x y z* `fill` *[physicalVolumeName]*", giving the position of a point in the target volume[3], the keyword `fill`, and optionally the name of the expected physical volume (used for verification purposes only). Only points actually in the physical volume will be generated, *no* points in daughter volumes will be generated. (This is so one can select the inner buffer and not get points in the scintillator, which is a daughter of the inner buffer.)
(3) "*x y z* `fill` *(physicalVolumeName* `|` `!`*) materialName*" specifies a volume to fill as above, but restricts the positions to those points within the physical volume *or daughters thereof* where the material is the one specified.
(4) "*x y z* `paint` *[physicalVolumeName [thickness [materialName]]]*" indicates surface-painting mode, optionally followed by the name of the physical volume expected at that position, optionally followed by the thickness of the coat of "paint" (positive puts the coat outside the volume, negative inside), optionally followed by the name of the material to which to restrict the "paint."

4 Description and user controls for vertex generators

There are only two styles of vertex generators currently implemented in KLG4sim: `KLVertexGen_Gun` and `KLVertexGen_HEPEvt`. The former is to provide a convenient, always-available source of particles for testing KLG4sim; the latter is the standard interface with external generator applications. Due to the division of labor between the position and vertex generators, the external generators usually don't need to know anything about the KamLAND geometry.

As with the position generators, the state of KLG4sim's internal vertex generators is set and read as a text string. **/generator/vtx/set *index [setting]*** is the fundamental command for controlling the state of the vertex generators. The *index* parameter is the

[3]This approach to specifying the volume isn't my favorite, but it is strongly motivated by subtleties in Geant4's geometry code.

index code from Table 3.[4] The *setting* parameter is the textual representation of the generator state, which generally must be enclosed in quotation marks. If the *setting* parameter is omitted, the command prints the current setting along with documentation on what the state string means for that generator.

KLVertexGen_Gun is used for the user-controlled test gun. Its state string has the format "*particleName dirX dirY dirZ kineticEnergyMeV [polX polY polZ]*". If *dirX = dirY = dirZ = 0*, the generated directions are isotropic; otherwise all particles are emitted in the given direction. If polarization has zero magnitude, a polarization is chosen randomly; if a particle has zero mass and spin one, the final polarization will be projected into the plane perpendicular to the momentum and made a unit vector.

KLVertexGen_HEPEvt is used for receiving events from external event generator applications. Its state string has the format "*filename*" or "*command* `|`", where the trailing Unix pipe symbol "`|`" causes the given command to be executed and input to be taken from the command's standard output. The format for KLHEPEvt ASCII data is described in the next section.

5 KLHEPEvt ASCII data format

`HEPEVT` is the name of a standard FORTRAN common block invented for high energy physics event generators some time ago. The current official documentation for `HEPEVT`, blessed by the Particle Data Group in the 2000 Review, is at
`http://www-pat.fnal.gov/stdhep.html`, which serves to define the concepts and variables used. Geant4 doesn't have any such common block, but the authors of Geant4 defined a basic way of interfacing such generators with Geant4 through a text file. Their prototypical example is given in Listing 1. I've extended the format and added some robustness and additional features to the input scanner, with advice from Yoshi Uchida and Bryan Tipton.[5]

KLHEPEvt files consist of comment lines, error lines, and event data. Comment lines are any line beginning with the "`#`" character or any blank line. Error lines are any line beginning with "`!`": they are just printed out on the Geant4 standard error stream whenever they are encountered. The format for KLHEPEvt event data is a line containing the number of particles in this event, followed by that many lines of particle data, as follows:

```
NHEP
ISTHEP IDHEP JDA1 JDA2 PX PY PZ PMASS DT0 DX0 DY0 DZ0 POLX POLY POLZ
ISTHEP IDHEP JDA1 JDA2 PX PY PZ PMASS DT0 DX0 DY0 DZ0 POLX POLY POLZ
ISTHEP IDHEP JDA1 JDA2 PX PY PZ PMASS DT0 DX0 DY0 DZ0 POLX POLY POLZ
...NHEP particle lines in all...
```

As a concrete example, an oxygen-13–nitrogen-13–carbon-13 decay sequence might look like this:

[4] This is also the code in the "vertex" column in Table 4 and the code in the `vtx` column of the output from the `/generator/list` command described previously.

[5] Any bugs or design errors are entirely my fault, of course.

```
4
1 -11 0 0  0.010000 0.000000 0.000000 0.000511  0.000000  # e+, 10 MeV/c, t=0
1  12 0 0 -0.008262 0.000000 0.000000 0.000000  0.000000  # nu, t=0
1  22 0 0  0.000000 0.003502 0.000000 0.000000  1.07e-29  # gamma, 3502 keV/c
1  12 0 0  0.000000 0.000000 0.002220 0.000000  5.94e+11  # nu, t=9.9min
```

The first 4 values on the "ISTHEP" lines are integers, and the remaining 11 values are real numbers (stored internally in double-precision). Any of the real numbers may be omitted. The real numbers may be used for communicating generator state by using special values of ISTHEP and/or IDHEP (see below). For a real particle to be tracked, ISTHEP should be 1 and IDHEP should be the PDG code of the particle.

Unfortunately, the PDG numbering scheme does not currently include nuclei, so a numbering scheme for nuclei has been invented for KLHEPEvt: a nuclei with atomic mass A and atomic number Z is encoded as IDHEP = $(9800000 + 1000 \times Z + A)$, giving 7-digit integers of the form `98zzaaa`.

JDA1, JDA2 are used for associating daughters with mother particles, which set "pre-assigned decay vertices" in Geant4, and are mostly useless and only occasionally harmless for us due to hidden assumptions in the Geant4 kernel. Talk to me if you want to use the mother-to-daughter associations.

ISTHEP is restricted to be in the range 1 to 203 (for technical reasons), and is interpreted as follows:

1	actual particle to be tracked by Geant4
2..99	informational particle, e.g., initial state particle (like a reactor antineutrino)
100..198	other information (generator state, radiative corrections, etc)
199	reserved for KLGeneratorAction
200..203	reserved just in case

If ISTHEP is less than 100, then the real numbers are converted from HEPEVT units (GeV/c, GeV/c^2, mm, ns[6]) to Geant4 internal units (MeV/c, MeV/c^2, mm, ns). If ISTHEP is greater than or equal to 100, then the numbers are stored internally with no unit conversions. I call these pseudo-particles "informatons." They are used for communicating generator-specific information, up to 11 quantities per informaton.

ISTHEP=199, IDHEP=-9999999 is used for a special informaton called the "global chronaton", with the following curious properties: its "X momentum" in Geant4 units is numerically equal to the universal time of the event (ns); its "Y momentum" in Geant4 units is numerically equal to the time since the last event (ns); its "Z momentum" in Geant4 units is numerically equal to the event type code (integer from Table 4).

Yoshi has also proposed that each generator send version information as well as any state information. I think this is good, although there is no reason to send it on every event. We might take ISTHEP=200 for this purpose, PX = vertex generator code (defined in Table 3), PY = major version number (changed only when interpretation of output data is affected), PZ = minor version number (changed when bug fixes or other upgrades made). Bryan uses ISTHEP=100, IDHEP=1,2,3 for generator file header, generator event header, and generator event checksum. The use of the informatons has not been completely standardized as of this writing (2001.08.31).

[6]N.B. The HEPEVT standard actually specifies mm/c for time, but the KLHEPEvt ASCII format stores time in ns.

A super-deluxe oxygen-13–nitrogen-13–carbon-13 decay sequence, including a checksum and extra information about initial, intermediate, and final nuclei, might look something like this:

```
8
3 9808013 0 0  0.000000  0.000000  0.000000 0         0.000000 # initial O-13
1     -11 0 0  0.010000  0.000000  0.000000 511e-6    0.000000 # e+
1      12 0 0 -0.008262  0.000000  0.000000 0.000000  0.000000 # nu
2 9807013 0 0 -0.001738  0.000000  0.000000 0         0.000000 # internal N-13
1      22 0 0  0.000000  0.003502  0.000000 0.000000  1.07e-29 # gamma
1      12 0 0  0.000000  0.000000  0.002220 0.000000  5.94e+11 # nu, t=9.9 min
1 9806013 0 0 -0.001738 -0.003502 -0.002220 0         5.94e+11 # final C-13
100     3 0 0 1188029421194.9988                               # checksum
```

A complete but trivial example of an external event generator is given in Listing 2.

Table 1: Global time generators $\{t_o\}$

Code	Description	Implemention
0	random activity	part of KLPrimaryGeneratorAction
1	accidental pile-up only	”
Total: 2		

Table 2: Global position generators $\{\vec{x}_o\}$

Code	Description	Implemention
0	uniform in scintillator volume	KLPosGen_PointPaintFill
1	uniform in cavern wall shell	”
2	uniform in ropes	”
3	uniform in bellmouth/chimney steel	”
4	uniform in balloon film	”
5	incident on balloon, z-dist. 1*	KLPosGen_ZDist+
6	incident on balloon, z-dist. 2*	”
7	incident on balloon, z-dist. 3*	”
8	uniform over area perpindicular to initial trajectory	KLPosGen_Cosmic
9	user-controlled	KLPosGen_PointPaintFill
10	spare user-controlled	”
11	uniform in inner buffer	”
Total: 12		

* “Z-distribution 1” has a probability in z suitable for gammas emitted near the spherical acrylic shell; “z-distribtution 2” is suitable for gammas emitted near the rock wall; “z-distribtution 3” is suitable for the overall flux of neutrons, pions, and so-forth from the wall (spallation products and (α, n) reactions). This is used for shell-based amplification of extremely low-probability events (codes 20, 21, and 22 in Table 4), as described in KamLAND-Note-01-02.
+ As of 2001-08-31, the external gamma generator actually provides the global positions of the incident gammas itself using its own geometry, so KLPosGen_null is used in KLG4sim instead of KLPosGen_ZDist in these slots.

Table 3: Vertex generators $\{\vec{p}_i, \delta\vec{x}_i, \delta t_i; i = 1..n\}$

Code	Description	Implemention	Author
0	reactor anti-neutrino	KLVertexGen_HEPEvt	Iwamoto, Uchida, Inoue, Vogel
1	terrestrial anti-neutrino	”	Enomoto
2	solar neutrino (incl. ^{8}B)	”	???
3	U $\rightarrow$ ^{222}Rn, sampled chain	KLVertexGen_HEPEvt or G4RadioactiveDecay	Piepke, Tipton
4	Th $\rightarrow$ ^{220}Rn, sampled chain	”	”
5	^{222}Rn $\rightarrow$^{206}Pb, entire chain	”	”
6	^{220}Rn $\rightarrow$^{208}Pb, entire chain	”	”
7	^{40}K	”	”

Table 3: (continued)

Code	Description	Implemention	Author
8	^{60}Co	”	”
9	^{85}Kr	”	”
10	^{10}C	”	”
11	^{11}C	”	”
12	^{14}C	”	”
13	cosmic-ray muons	KLVertexGen_HEPEvt	Maricic
14	external gammas on balloon from material in tank	”	Tipton, Hoffman
15	external gammas on balloon from water and rocks	”	”
16	spallation products and neutrons on balloon from rocks	”	Young, Detwiler, Wang
17	user-controlled test gun	KLVertexGen_Gun	Horton-Smith
18	user-controlled HEPEvt	KLVertexGen_HEPEvt	Horton-Smith
Total: 19			

Table 4: Primary event generators

Code	Description	Sub-generator code		
		time	position	vertex
0	Reactor anti-neutrino	0	0	0
1	Terrestrial anti-neutrino	0	0	1
2	Solar neutrino	0	0	2
3	Particle gun	0	9	17
4	Rope Uranium	0	2	3
5	Rope Thorium	0	2	4
6	Rope Radon	0	2	5
7	Rope Thoron (^{220}Rn)	0	2	6
8	Rope ^{40}K	0	2	7
9	Bellmouth/chimney Uranium	0	3	3
10	Bellmouth/chimney Thorium	0	3	4
11	Bellmouth/chimney Radon	0	3	5
12	Bellmouth/chimney Thoron	0	3	6
13	Bellmouth/chimney ^{40}K	0	3	7
14	Bellmouth/chimney ^{60}Co	0	3	8
15	Balloon film Uranium	0	4	3
16	Balloon film Thorium	0	4	4
17	Balloon film Radon	0	4	5
18	Balloon film Thoron	0	4	6
19	Balloon film ^{40}K	0	4	7
20	Buffer-to-balloon gammas	0	5	14
21	OD-to-balloon gammas	0	6	15
22	Wall-to-balloon hadrons	0	7	16
23	Cosmic muons	0	8	13
24	Scintillator Uranium	0	0	3

Table 4: (continued)

Code	Description	Sub-generator code		
		time	position	vertex
25	Scintillator Thorium	0	0	4
26	Scintillator Radon	0	0	5
27	Scintillator Thoron	0	0	6
28	Scintillator ^{40}K	0	0	7
29	Scintillator ^{85}Kr	0	0	9
30	Scintillator ^{10}C	0	0	10
31	Scintillator ^{11}C	0	0	11
32	Scintillator ^{14}C	1	0	12
33	Dome/wall Uranium	1	1	3
34	Dome/wall Thorium	1	1	4
35	Dome/wall Radon	1	1	5
36	Dome/wall Thoron	1	1	6
37	Dome/wall ^{40}K	1	1	7
38	Spare KLHEPEvt	0	10	18
39	Inner buffer reactor antinu	0	11	0
40	Inner buffer geo-antinu	0	11	1
41	Inner buffer solar neutrino	0	11	2
42	Inner buffer Uranium	0	11	3
43	Inner buffer Thorium	0	11	4
44	Inner buffer Radon	0	11	5
45	Inner buffer Thoron	0	11	6
46	Inner buffer ^{40}K	0	11	7
47	Inner buffer ^{85}Kr	0	11	9
48	Inner buffer ^{10}C	0	11	10
49	Inner buffer ^{11}C	0	11	11
50	Inner buffer ^{14}C	0	11	12
Total: 51				

Listing 1: Definition and example of values passed to G4HEPEvtInterface.

```
************************************************************
      SUBROUTINE HEP2G4
*
* Convert /HEPEVT/ event structure to an ASCII file
* to be fed into G4HEPEvtInterface.
*
* [Reminder: all that matters to G4HEPEvtInterface is the data in
* the ASCII file: you really don't have to use a HEPEVT common
* block in your own program if you don't want to.  Also note that
* many fields in HEPEVT are not actually used.  In particular,
* the information in VHEP is lost, which is a pain.]
*
* More documentation on G4HEPEvtInterface can be found at
*   http://wwwinfo.cern.ch/asd/geant4/G4UsersDocuments/UsersGuides/
*     ForApplicationDeveloper/html/PracticalApplications/
*       eventGenerator.html#3.5.2
*
************************************************************
      PARAMETER (NMXHEP=2000)
      COMMON/HEPEVT/NEVHEP,NHEP,ISTHEP(NMXHEP),IDHEP(NMXHEP),
     >JMOHEP(2,NMXHEP),JDAHEP(2,NMXHEP),PHEP(5,NMXHEP),VHEP(4,NMXHEP)
      DOUBLE PRECISION PHEP,VHEP
*
*  NEVHEP              =  The event number
*  NHEP                =  The number of particles in this event
*  ISTHEP(NMXHEP)      =  The Particle status (1 for primary)
*  IDHEP(NMXHEP)       =  The particle PDG code
*  JMOHEP(2,NMXHEP)    =  Indices of particle's first and last mother
*  JDAHEP(2,NMXHEP)    =  Indices of particle's first and last daughter
*  PHEP(5,NMXHEP)      =  4-Momentum, mass (GeV/c, GeV, GeV/c**2)
*  VHEP(4,NMXHEP)      =  Vertex information: position(mm), time(mm/c)
*
      WRITE(6,*) NHEP
      DO IHEP=1,NHEP
       WRITE(6,10)
     >  ISTHEP(IHEP),IDHEP(IHEP),JDAHEP(1,IHEP),JDAHEP(2,IHEP),
     >  PHEP(1,IHEP),PHEP(2,IHEP),PHEP(3,IHEP),PHEP(5,IHEP)
10     FORMAT(4I5,4(1X,D15.8))
      ENDDO
*
      RETURN
      END
```

Listing 2: Example of complete event generator

```
// trivial_muon_demo.cc
//
// compile using
//   ($CXX)  trivial_muon_demo.cc  -lCLHEP  -o trivial_muon_demo
//
// This is primarily to demonstrate how to write out events in
// HEPEVT ASCII format.
//
// It generates tracks with a cos(theta) distribution of directions,
// gives them a kinetic energy distributed according to (dN/dE)=(E+E0)**(-g-1),
// flips an equally-weighted coin to decide mu+ or mu-,
// and prints out the event in the proper format.
//

#include <stdlib.h>
#include <iostream>
#include <CLHEP/Vector/ThreeVector.h>
#include <CLHEP/Random/Randomize.h>

using namespace std;

Hep3Vector RandCosDist();

int main(int argc, char **argv)
{
  const double E0_GeV= 500.0;
  const double gamma= 2.70;
  int nevent=1000000;

  // print out introductory information
  cout << "# trivial_muon_demo, initial version\n"
       << "# E0=" << E0_GeV << " GeV, gamma=" << gamma << endl;

  for (int ievent=0; ievent<nevent; ievent++) {

    double E_GeV= ( pow( 1.0-HepUniformRand(), -1.0/gamma ) - 1.0 ) * E0_GeV;

    Hep3Vector momentum_GeV = - E_GeV * RandCosDist();  // muon mass ignored.

    int PDGcode;
    if (HepUniformRand() >= 0.5)
      PDGcode=  13; // mu-
    else
      PDGcode= -13; // mu+ (antimu-)
```

```
    // First, a line for the number of particles in this event.
    cout << 1 << endl;                    // NHEP -- always 1 for us

    // Next, line(s) for each particle.  Don't forget spaces between fields.
    cout << 1 << ' '                      // ISTHEP -- 1 for real particles
         << PDGcode << ' '                // IDHEP  -- the PDG code
         << 0 << ' ' << 0 << ' '          // daughter indices
         << momentum_GeV.x() << ' '
         << momentum_GeV.y() << ' '
         << momentum_GeV.z() << endl;
  }

  return 0;
}


Hep3Vector RandCosDist()
{
  // the following cute sequence generates a cos(theta) distribution!
  double u,v,w;
  do {
    u= HepUniformRand()*2.0-1.0;
    v= HepUniformRand()*2.0-1.0;
    w= 1.0- (u*u+v*v);
  } while (w < 0.0);
  w= sqrt(w);

  return Hep3Vector(u,v,w);
}
```