LaTeX-ed on March 29, 2024.

HepMC 2

a C++ Event Record for Monte Carlo Generators

http://savannah.cern.ch/projects/hepmc/

User Manual Version 2.06
May 17, 2010

Matt Dobbs
University of Victoria, Canada

Jørgen Beck Hansen
CERN

Lynn Garren
Fermi National Accelerator Laboratory

Lars Sonnenschein
RWTH Aachen

Abstract

The HepMC package is an object oriented event record written in C++ for High Energy Physics Monte Carlo Generators. Many extensions from HEPEVT, the Fortran HEP standard, are supported: the number of entries is unlimited, spin density matrices can be stored with each vertex, flow patterns (such as color) can be stored and traced, integers representing random number generator states can be stored, and an arbitrary number of event weights can be included. Particles and vertices are kept separate in a graph structure, physically similar to a physics event. The added information supports the modularisation of event generators. The package has been kept as simple as possible with minimal internal/external dependencies. Event information is accessed by means of iterators supplied with the package.

HepMC 2 is an extension to the original HepMC written by Matt Dobbs.

Contents

1 Introduction **2**
1.1 Features of the HepMC Event Record 3

2 HepMC 2 **3**
2.1 Overview of Changes Since HepMC 1.26 4

3 Package Structure **4**
3.1 Dependencies 7
3.2 Namespace 7
3.3 Performance 7

4 Overview of Core Classes **8**
4.1 HepMC::GenEvent 8
4.1.1 HepMC::PdfInfo 9
4.1.2 HepMC::HeavyIon 10
4.1.3 HepMC::GenCrossSection 10
4.1.4 HepMC::Units 10
4.2 HepMC::GenVertex 11
4.3 HepMC::WeightContainer 11
4.4 HepMC::GenParticle 12
4.4.1 HepMC::Flow 13
4.4.2 HepMC::Polarization 13
4.4.3 HepMC::FourVector 13
4.5 HepMC::IO_BaseClass 14

5 Overview of Iterators **14**
5.1 HepMC::GenEvent::vertex_iterator 15
5.2 HepMC::GenEvent::vertex_const_iterator 15
5.3 HepMC::GenEvent::particle_iterator 15
5.4 HepMC::GenEvent::particle_const_iterator 15
5.5 HepMC::GenVertex::vertex_iterator 15
5.6 HepMC::GenVertex::particle_iterator 16

6 Ascii Output **17**
6.1 Basic IO_GenEvent Structure 17
6.2 General Event Information 19
6.3 Vertices and Particles 20

7 Building HepMC **20**

8 Examples **21**

9 Deprecated Classes **24**

10 Acknowlegements **24**

List of Figures

1 Modularised event generation 2
2 Event visualization 3
3 HepMC class diagrams 6
4 Example of ascii format 18
5 Example of ascii format 18
6 Example of ascii format 18
7 IO_GenEvent example 21
8 Streaming input example 22
9 IO_HEPEVT and IO_GenEvent example 22
10 IO_HEPEVT and streaming output example 23

List of Tables

1 Performance 7

1 Introduction

This user manual is intended as a companion to the online documentation[1], and together with the examples should provide a friendly introduction to the HepMC event record. A general overview is available in Ref. [1].

The HEP community has moved towards Object-Oriented computing tools (usually C++): most upcoming experiments are choosing OO software architecture, and Pythia 8 [2] and Herwig++ [3], written in C++, are available. A standard event record must be simple for the end user to access information, while maintaining the power and flexibility offered by OO design. The HepMC event record has been developed to satisfy these criteria.

HepMC is an object oriented event record written in C++ for Monte Carlo Generators in High Energy Physics. It has been developed independent of a particular experiment or event generator. It is intended to serve as both a "container class" for storing events after generation and also as a "framework" in which events can be built up inside a set of generators. This allows for the modularisation of event generators—wherein different event generators could be employed for different steps or components of the event generation process (illustrated in Figure 1). [2]

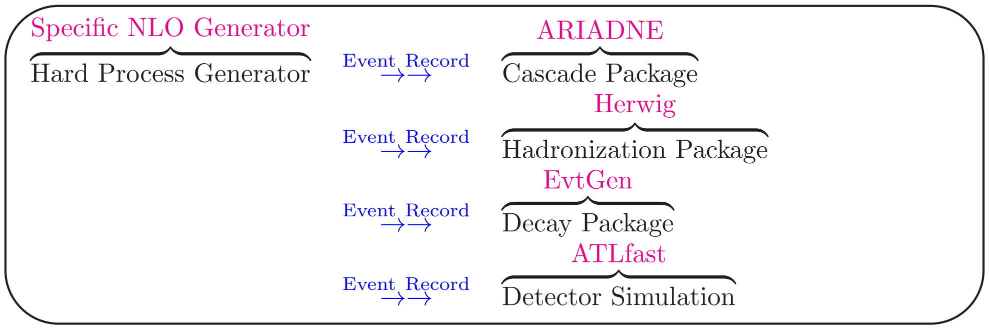


Figure 1: HepMC supports the concept of modularised event generation (illustrated above) by containing sufficient information within the event record to act as a messenger between two modular steps in the event generation process.

Physics events are generally visualised using diagrams with a graph structure (Figure 2, left) which HepMC imitates by separating out particles from vertices and representing them as the edges and nodes respectively of a graph [3] (Figure 2, right). Each vertex maintains a listing of its incoming and outgoing particles, while each particle points back to its production vertex and decay vertex. The extension to multiple collisions is natural - the super-position of graphs from several different initial processes - and so the event may contain an unlimited number of (possibly interconnected) graphs. The number of vertices/particles in each event is also open-ended. A subset of the event (such as one connected graph or a single vertex and its descendants) may be examined or modified

[1]http://lcgapp.cern.ch/project/simu/HepMC/

[2]At the *Physics at TeV Colliders Workshop 2001* in Les Houches, France, a group of Monte Carlo authors and experimentalists produced a document [4] which outlines the information content necessary for two event generators to communicate information about a hard process to the subsequent stages of event generation. This was implemented in a set of Fortran common blocks, and many ideas from HepMC were used, such as the scheme for handling color flow information. Version 1.1 of HepMC supports the full event information content of Ref. [4] (run information—pertaining to a collection of events—is also specified in that document and is not addressed in HepMC).

[3]Ref. [5] uses a similar structure.

without having to interpret complex parent/child relationship codes or re-shuffle the rest of the event record.

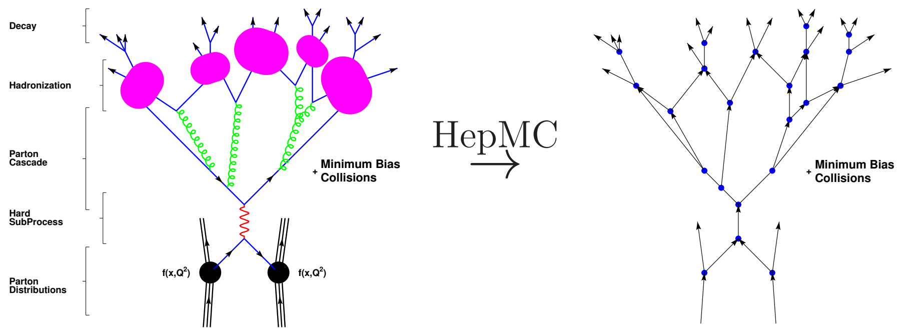


Figure 2: Events in HepMC are stored in a graph structure (right), similar to a physicist's visualisation of a collision event (left).

1.1 Features of the HepMC Event Record

- simple - easy access to information provided by iterators and range methods
- minimum dependencies
- information is stored in a graph structure, physically similar to a collision event
- allows specification of momentum and length units
- allows for the inclusion of spin density matrices
- allows for the tracing of an arbitrary number of flow patterns
- ability to store the state of random number generators (as integers)
- ability to store an arbitrary number of event weights
- ability to store parton distribution function information
- ability to store heavy ion information
- ability to store generated cross section information on an event by event basis
- strategies for conversion to/from HEPEVT (Ref. [4]) which are easily extendible to support other event records
- strategies for input/output to/from Ascii files which are easily extendible to support other forms of persistency
- support for standard streaming I/O

2 HepMC 2

Since January 2006, HepMC has been supported as an LCG external package. The official web site is now http://savannah.cern.ch/projects/hepmc/, and compiled libraries for supported platforms are available at /afs/cern.ch/sw/lcg/external/HepMC.

Historically, HepMC has used CLHEP (Ref. [6]) Lorentz vectors. Some users wished to use a more modern Lorentz vector package. At the same time, there was concern about allowing dependencies on any external package. Therefore, the decision was made to replace the CLHEP Lorentz vectors with a minimal vector representation within HepMC.

Because this is a major change, the versioning was changed from 1.xx.yy to 2.xx.yy. Normally, a version number change in *xx* represents a change to the code and a version number change in *yy* represents a bug fix.

There have also been continuing requests for other features. Changes to HepMC must be approved by the LCG simulation project (Ref. [7]).

2.1 Overview of Changes Since HepMC 1.26

See the HepMC ChangeLog [8] for a complete listing.

GenEvent now contains pointers to a heavy ion class, a PDF information class, and a generated cross section class. The pointers are null by default. In addition, GenEvent now has the capability to declare which momentum and position units are used.

GenParticle momentum and GenVertex position are represented by a simple FourVector class instead of the CLHEP Lorentz vectors. The SimpleVector.h header contains the FourVector and ThreeVector classes. GenVertex will return the ThreeVector portion of the position. Polarization will accept or return a ThreeVector representation of the polarization.

Both FourVector and ThreeVector have templated constructors. These constructors allow you to use the GenParticle and GenVertex constructors with *any* Lorentz vector, as long as the Lorentz vector has `x()`, `y()`, `z()`, and `t()` methods.

The generated mass, which has always been part of the HEPEVT common block, is now stored in GenParticle. When a particle has large momentum and small mass, calculating the mass from the momentum is unreliable. Also, different machine representations and roundoff errors mean that a calculated mass is not always consistent. If no generated mass is set, then the mass is calculated from the momemementum and stored in GenParticle.

The IO_AsciiParticles class provides output in the Pythia style. This output is intended for ease of reading event output, not for persistency.

The IO_Ascii output class has been replaced with of IO_GenEvent, described in Section 6. IO_GenEvent persists all information in the updated GenEvent object and uses iostreams for greater flexibility. IO_GenEvent also has a constructor taking a file name and mode type for backwards compatibility. Output remains in Ascii format.

Streaming I/O of GenEvent objects may also be done without using IO_GenEvent.

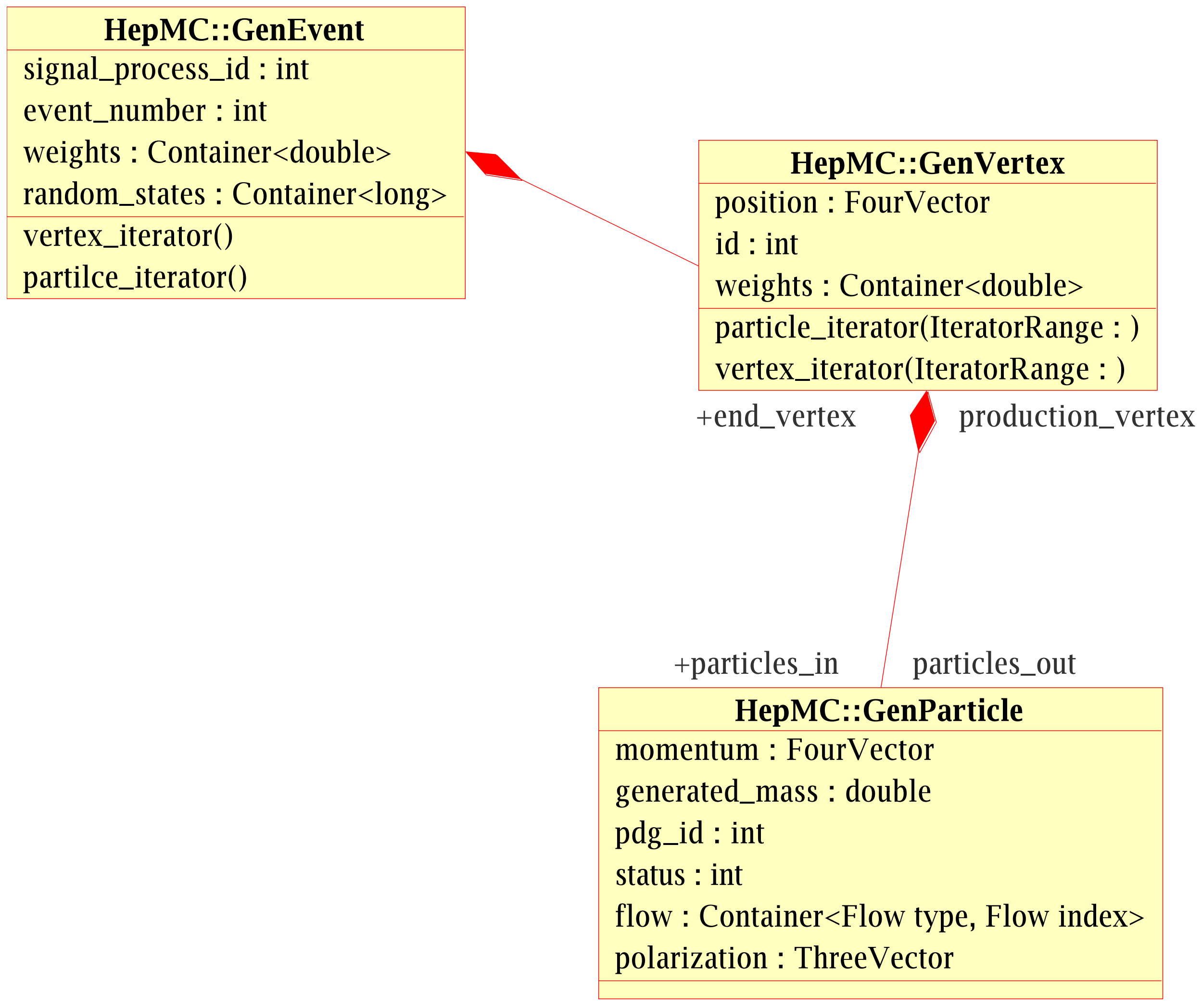
The new HepMCDefs.h header allows users to query which features are enabled.

3 Package Structure

Entries within the event record are separated into particles and vertices. Each particle is composed of momentum, flow, and polarization classes as well as id and status information. The vertices are the connection nodes between particles and are a container class for particles: thus each particle within an event belongs to at least one vertex. In addition the vertex is composed of position, id, and (spin density matrix) weight information. Particles and vertices are uniquely identified by an integer—referred to as a "barcode"—which is meant to be a persistent label for a particular particle instance. The event is the container class for all (possibly inter-connected) vertices in the event and contains process id, event number, unit information, weight, and random number state information.

Please note that the barcodes are intended for internal use within HepMC as a unique identifier for the particles and vertices. Using the barcode to encode extra information is an abuse of the barcode data member and causes confusion among users.

Iterators are provided as methods of the vertex and event classes which allow easy directed access to information in the C++ Standard Template Library (STL) style. In addition, the GenRanges header contains several iterator classes which explicity provide begin() and end() methods for the various public iterator types, allowing the user to easily use such utilities as std::foreach.



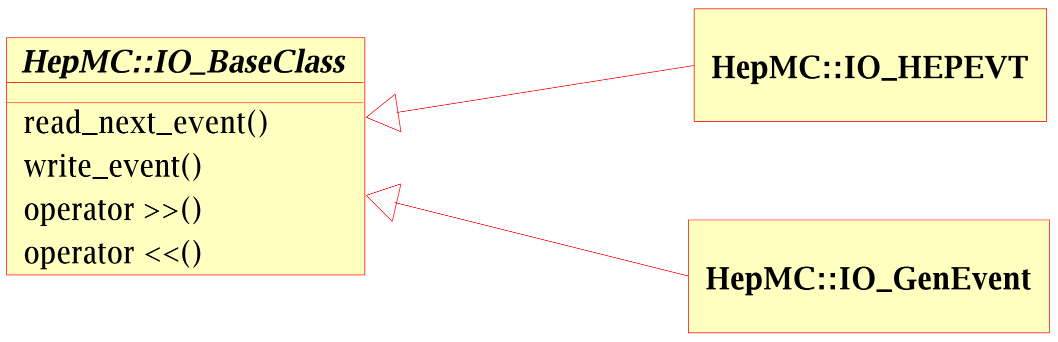


Figure 3: Class diagrams for the event record classes (GenEvent, GenVertex, and GenParticle) and the IO strategies are shown in the UML notation.

The event record class relationships are shown in Figure 3.

Several input/output strategies are provided. The interface for these strategies is defined by an abstract base class, IO_BaseClass. These strategies are capable of input/output of events and as such they depend directly on the event record class.

The package consists of about 5500 lines of code including ≈1500 comments. There are 7 core classes (GenEvent, GenVertex, GenParticle, Flow, Polarization, WeightContainer, IO_BaseClass) and several utility classes (i.e. IO_HEPEVT, IO_GenEvent, HEPEVT_Wrapper, PythiaWrapper, ...).

3.1 Dependencies

The HepMC 2 package depends only on the C++ Standard Template Library [9] (STL).

The HepMC 1 package dependencies were limited to STL and the vector classes from the Class Library for High Energy Physics [6] (CLHEP).

Simple wrappers for the Fortran versions of Pythia [10] and Herwig [11] are supplied with the package to allow the inclusion of event generation examples.

3.2 Namespace

The HepMC package is written within the HepMC:: namespace. Outside of this namespace all class names will need to be prefixed by HepMC::.

The units methods and enums are in the HepMC::Units:: namespace.

3.3 Performance

The CPU time performance of the HepMC event record has been quantified by generating 1000 LHC $W\gamma$ production events using Pythia 5.7 and transferring the event record to HepMC using the IO_HEPEVT strategy. Results are summarised in Table 1. Generation of events in Pythia required 29 seconds of CPU time. Generating the same events and transferring them into HepMC required 34 seconds.

	CPU Time	File Size
Pythia	29 sec	
+ HepMC	34 sec	
+ HepMC + IO_Ascii	64 sec	60.5 Mbytes
+ LCWRITE	92 sec	106 Mbytes
HEPEVT raw size 1K events, 500K particles		48.2 Mbytes

Table 1: CPU time performance and file size using a dedicated 450 MHz Pentium III.

The time to write HepMC events as Ascii files using the IO_Ascii strategy was compared to `LCWRITE`, a simple Fortran routine used in NLC studies to write the HEPEVT common block in formatted Ascii to file. Generating the events with Pythia, transferring them to HepMC, and writing them to file took 64 seconds and produced a 60.5 Mbyte file. Generating the events with Pythia and writing them to file using the `LCWRITE` subroutine took 92 seconds and produced a 106 Mbyte file. Compression algorithms (such as `gzip`) can reduce the file sizes by a factor 3 or more. The raw size of the HEPEVT common block for these 1000 events (which in this case produced about 500K particles) is 48.2 Mbytes. In both cases most CPU time is spent writing to file. HepMC benefits from added logic when interpreting the record and from position information which is stored once for each vertex, rather than with every particle.

CPU time savings will be realized when HepMC is used inside event generators - since it is possible to target and modify one area of the particle/vertex graph without re-shuffling the rest of the event record.

4 Overview of Core Classes

NOTE ABOUT UNITS: HepMC does not define which units are used for the information stored in the event record. The HEPEVT standard uses GeV/mm, and so the output from most Fortran generators will normally be in these units. CLHEP and Geant4 use MeV/mm, and some collaborations such as ATLAS have adopted these units for their simulation. We also note that Fluka uses cm.

As of HepMC 2.04.00, GenEvent contains member data to store information about units used. Default units are declared with configure switches –with-momentum and –with-length. These configure switches are required.

HepMC users should refer to the code that fills the event record to determine which units are being used.

4.1 HepMC::GenEvent

Important Public Methods

- **GenEvent(Units::MomentumUnit, Units::LengthUnit,...)**: *As of HepMC 2.04.00, additional constructors are available to enable unit specification - see also use_units below.*
- **add_vertex**: *adopts the specified vertex to the event and assumes responsibility for deleting the vertex*
- **remove_vertex**: *removes the specified vertex from the event, the vertex is not deleted - thus use of this method is normally followed directly by a delete vertex operation*
- **vertex_iterator**: *iterates over all vertices in the event - described in the iterator section*
- **particle_iterator**: *iterates over all particles in the event - described in the iterator section*
- **vertex_const_iterator**: *constant version of the vertex_iterator*
- **particle_const_iterator**: *constant version of the particle_iterator*
- **vertex_range**: *provides begin() and end() for ease of use with external fuctions, such as for_each*
- **particle_range**: *provides begin() and end() for ease of use with external fuctions, such as for_each*
- **print**: *gives a formatted printout of the event to the specified output stream*
- **barcode_to_particle**: *returns a pointer to the particle associated with the integer barcode argument*
- **barcode_to_vertex**: *returns a pointer to the vertex associated with the integer barcode argument*
- **use_units**: *set both momentum and position units, and scale FourVectors if necessary*
- **read**: *read ascii input directly from an istream*
- **write**: *write ascii output directly to an ostream*

Relevant Data Members

- **signal_process_id**: *an integer ID uniquely specifying the signal process (i.e. MSUB in Pythia).*
- **event_number**: *integer*
- **event_scale**: *(optional) the scale of this event. (-1 denotes unspecifed)*
- **alphaQCD**: *(optional) the value of the strong coupling constant α_{QCD} used for this event. (-1 denotes unspecifed)*
- **alphaQED**: *(optional) the value of the electroweak coupling constant α_{QED} (e.g. $\frac{1}{128}$) used for this event. (-1 denotes unspecifed)*
- **signal_process_vertex**: *(optional) pointer to the vertex defined as the signal process - allows fast navigation to the core of the event*
- **beam_particle_1**: *(optional) pointer to the first incoming beam particle*

- **beam_particle_2**: *(optional) pointer to the second incoming beam particle*
- **weights**: *a container of an arbitrary number of 8 byte floating point event weights and their associated names*
- **random_states**: *a container of an arbitrary number of 4 byte integers which define the random number generator state just before the event generation*
- **heavy_ion**: *(optional) a pointer to a HeavyIon object (zero by default)*
- **pdf_info**: *(optional) a pointer to a PdfInfo object (zero by default)*
- **cross_section**: *(optional) a pointer to a GenCrossSection object (zero by default)*
- **momentum_unit**: *momentum units (MEV or GEV)*
- **length_unit**: *position units (MM or CM)*

IMPORTANT FREE FUNCTIONS

- **operator** >>: *read ascii input directly from an istream*
- **operator** <<: *write ascii output directly to an ostream*

NOTES AND CONVENTIONS

- if hit and miss Monte Carlo integration is to be performed with a single event weight, the first weight will be used by default
- Memory allocation: vertex and particle objects will normally be created by the user with the NEW operator. Once a vertex (particle) is added to a event (vertex), it is "adopted" and becomes the responsibility of the event (vertex) to delete that vertex (particle).
- Although default units are specified with configure, the user is strongly encouraged to explicitly set units with either the appropriate constructor or a call to set_units.

The GenEvent is the container class for vertices. A listing of all vertices is maintained with the event, giving fast access to vertex information. GenParticles are accessed by means of the vertices.

Extended event features (weights, random_states, heavy_ion, pdf_info, cross_section) have been implemented such that if left empty/unused performance and memory usage will be similar to that of an event without these features.

Iterators are provided as members of the GenEvent class (described in Section 5). Methods which fill containers of particles or vertices are *not provided*, as the STL provides these functionalities with algorithms such as `copy` and iterator adaptors such as `back_inserter` giving a clean generic approach. Using this functionality it is easy to obtain lists of particles/vertices given some criteria - such as a list of all final state particles. Classes which provide the criteria (called predicates) are also *not provided*, as the number of possibilities is open ended and specific to the application - and would clutter the HepMC package. Implementing a predicate is simple (about 4 lines of code). Examples are given in the GenEvent header file and in `example_UsingIterators.cc` (Section 8). Useful examples can also be found in testHepMCIteration.cc.

The signal_process_id is packaged with each event (rather than being associated with a run class for example) to handle the possibility of many processes being generated within the same run. A container of tags specifying the meaning of the weights and random_states entries is envisioned as part of a run class - which is beyond the scope of an event record.

4.1.1 HepMC::PdfInfo

RELEVANT DATA MEMBERS

- **id1**: *flavour code of first parton*
- **id2**: *flavour code of second parton*
- **pdf_id1**: *LHAPDF set id of first parton*
- **pdf_id2**: *LHAPDF set id of second parton*
- **x1**: *fraction of beam momentum carried by first parton ("beam side")*
- **x2**: *fraction of beam momentum carried by second parton ("target side")*
- **scalePDF**: *Q-scale used in evaluation of PDF's (in GeV)*

- **pdf1**: *PDF (id1, x1, Q) This should be of the form x*f(x)*
- **pdf2**: *PDF (id2, x2, Q) This should be of the form x*f(x)*

<u>NOTES AND CONVENTIONS</u>

- The LHAPDF [12] set ids are the entries in the first column of http://projects.hepforge.org/lhapdf/PDFsets.index.
- The LHAPDF set ids pdf_id1 and pdf_id2 are zero by default.
- IMPORTANT: The contents of pdf1 and pdf2 are expected to be x*f(x), which is the quantity returned by LHAPDF.
- IMPORTANT: Input parton flavour codes id1 and id2 are expected to obey the PDG code conventions, especially g = 21.

PdfInfo stores additional PDF information for a GenEvent. Creation and use of this information is optional.

4.1.2 HepMC::HeavyIon

<u>RELEVANT DATA MEMBERS</u>

- **Ncoll_hard**: *Number of hard scatterings*
- **Npart_proj**: *Number of projectile participants*
- **Npart_targ**: *Number of target participants*
- **Ncoll**: *Number of NN (nucleon-nucleon) collisions*
- **N_Nwounded_collisions**: *Number of N-Nwounded collisions*
- **Nwounded_N_collisions**: *Number of Nwounded-N collisons*
- **Nwounded_Nwounded_collisions**: *Number of Nwounded-Nwounded collisions*
- **spectator_neutrons**: *Number of spectators neutrons*
- **spectator_protons**: *Number of spectators protons*
- **impact_parameter**: *Impact Parameter(fm) of collision*
- **event_plane_angle**: *Azimuthal angle of event plane*
- **eccentricity**: *eccentricity of participating nucleons in the transverse plane (as in phobos nucl-ex/0510031)*
- **sigma_inel_NN** : *nucleon-nucleon inelastic (including diffractive) cross-section*

HeavyIon provides additional information storage in GenEvent for Heavy Ion generators. Creation and use of this information is optional.

4.1.3 HepMC::GenCrossSection

<u>RELEVANT DATA MEMBERS</u>

- **cross_section**: *cross section in pb*
- **cross_section_error**: *error associated with this cross section in pb*

GenCrossSection provides additional storage in GenEvent for an event by event snapshot of the cross section while events are being generated. It is expected that the final cross section will be stored elsewhere. Creation and use of this information is optional.

4.1.4 HepMC::Units

<u>IMPORTANT METHODS</u>

- **enum MomentumUnit**: *values are MEV or GEV*
- **enum LengthUnit**: *values are MM or CM*
- **std::string name(MomentumUnit)**: *return the unit designation as a string*
- **std::string name(LengthUnit)**: *return the unit designation as a string*

<u>NOTES AND CONVENTIONS</u>

- Refer to a unit enum as, for instance, HepMC::Units::GEV
- Whenever both unit types are passed, MomentumUnit always goes first.

Units is a namespace encapsulating methods used for unit manipulation. Default units are set at compile time by the configure switches –with-momentum and –with-length.

4.2 HepMC::GenVertex

Important Public Methods

- **add_particle_in**: *adds the specified particle to the container of incoming particles*
- **add_particle_out**: *adds the specified particle to the container of outgoing particles*
- **remove_particle**: *removes the specified particle from both/either of the incoming/outgoing particle containers, the particle is not deleted - thus use of this method is normally followed directly by a delete particle operation*
- **vertex_iterator**: *iterates over vertices in the graph, given a specified range - described in the iterator section*
- **particle_iterator**: *iterates over particles in the graph, given a specified range - described in the iterator section*
- **particles**: *provides begin() and end() for ease of use with external fuctions, such as for_each*
- **particles_in**: *provides begin() and end() for ease of use with external fuctions, such as for_each*
- **particles_out**: *provides begin() and end() for ease of use with external fuctions, such as for_each*

Relevant Data Members

- **position**: $\vec{x}, ct$ *stored as FourVector*
- **id**: *integer id, may be used to specify a vertex type*
- **weights**: *a container of 8 byte floating point numbers of arbitrary length, could be mapped in pairs into rows and columns to form spin density matrices of complex numbers*
- **barcode**: *an integer which uniquely identifies the GenVertex within the event. For vertices the barcodes are always negative integers.*

Notes and Conventions

- no standards are currently defined for the vertex id
- once a particle is added, the vertex becomes its owner and is responsible for deleting the particle

The GenVertex is the container class for particles and forms the nodes which link particles into a graph structure.

The weights container is included with each vertex with the intention of storing spin density matrices. It is envisioned that a generator package would assign spin density matrices to particle production vertices and provide the functional form of the frame definition for the matrix as a "look-up" method for interpreting the weights. The generator package would also provide a boost method to go from the frame of the density matrix to the lab frame and back without destroying correlations. This gives maximum freedom to the sub-generators - allowing for different form definitions. This implementation is consistent with the EvtGen B-decay package [13] requirements.

4.3 HepMC::WeightContainer

Relevant Data Members

- **weights**: *a vector of 8-byte floating point weights*
- **names**: *a map of strings associated with a position in the weights vector*

Notes and Conventions

- methods are coded and names chosen in the spirit of the STL vector class

The WeightContainer is a storage area for double precision weights used in GenEvent and GenVertex and their associated names. WeightContainer mimics both a map class and the STL vector class, and its member functions are chosen in that spirit. You might, for instance, use the GenEvent weights to include information about differential cross sections.

4.4 HepMC::GenParticle

<u>Important Public Methods</u>

- **operator FourVector**: *conversion operator - resolves the particle as a 4-vector according to its momentum*
- **generatedMass**: *generated mass*
- **momentum().m()**: *calculates mass from momentum*
- **particles_in**: *provides begin() and end() for ease of use with external fuctions, such as for_each*
- **particles_out**: *provides begin() and end() for ease of use with external fuctions, such as for_each*

<u>Data Members</u>

- **momentum**: $\vec{p}, cE$ *stored as FourVector*
- **generated_mass**: *generated mass for this particle*
- **pdg_id**: *unique integer ID specifying the particle type*
- **status**: *integer specifying the particle's status (i.e. decayed or not)*
- **flow**: *allows for the storage of flow patterns (i.e. color flow), refer to Flow class*
- **polarization**: *stores the particle's polarization as* (θ, ϕ)*, refer to Polarization class*
- **production_vertex**: *pointer to the vertex where the particle was produced, can only be set by the vertex*
- **end_vertex**: *pointer to the vertex where the particle decays, can only be set by the vertex*
- **barcode**: *an integer which uniquely identifies the GenParticle within the event. For particles the barcodes are always positive integers.*

<u>Notes and Conventions</u>

- the particle ID should be specified according to the PDG standard [14]
- the status code should be specified according to the clarified HEPEVT status codes

The particle is the basic unit within the event record. The GenParticle class is composed of the FourVector, Flow, and Polarization classes.

Pointers to the particle's production and end vertex are included. In order to ensure consistency between vertices/particles - these pointers can only be set from the vertex. Thus adding a particle to the particles_in container of a vertex will automatically set the end_vertex of the particle to point to that vertex.

The definition of a HepLorentzVector scope resolution operator allows for the use of 4-vector algebra with particles (i.e. preceding an instance, `particle`, of the HepMC::GenParticle class by `(HepLorentzVector)particle` causes it to behave exactly like its 4-vector momentum, examples are given in the particle header file).

A second 4-vector for the particle's momentum at decay time has *not* been included (as for example in [5], where the second momentum vector is included to facilitate tracking through material). If this is desirable, one can simply add a decay vertex with the same particle type going out. This is intuitive, since a change in momentum cannot occur without an interaction (vertex).

After some discussion, the authors in MCnet [18] have agreed to a clarification of the HEPEVT [15] status codes. The Fortran Monte Carlo generators will not change their behaviour, but Sherpa, Pythia8, and Herwig++ will go to the newer usage.
These are the accepted status code definitions:

- **0** : *an empty entry with no meaningful information and therefore to be skipped unconditionally*
- **1** : *a final-state particle, i.e. a particle that is not decayed further by the generator (may also include unstable particles that are to be decayed later, as part of the detector simulation). Such particles must always be labelled '1'.*
- **2** : *decayed Standard Model hadron or tau or mu lepton, excepting virtual intermediate states thereof (i.e. the particle must undergo a normal decay, not e.g. a shower branching). Such particles must always be labelled '2'. No other particles can be labelled '2'.*
- **3** : *a documentation entry*
- **4** : *an incoming beam particle*

- **5-10** : *undefined, reserved for future standards*
- **11-200**: *an intermediate (decayed/branched/...) particle that does not fulfill the criteria of status code 2, with a generator-dependent classification of its nature.*
- **201-** : *at the disposal of the user, in particular for event tracking in the detector*

4.4.1 HepMC::Flow

Important Public Methods
- **connected_partners**: *returns a container of all particles connected via the specified flow pattern*
- **dangling_connected_partners**: *returns a container of all particles "dangling" from the ends of the specified flow pattern*

Relevant Data Members
- **particle_owner**: *points back to the particle to which the flow object belongs*
- **icode map**: *container of integer flow codes - each entry has an index and an icode*

Notes and Conventions
- code indices 1 and 2 are reserved for color flow

The Flow class is a data member of the GenParticle—its use is optional. It stores flow pattern information as a series of integer flow codes and indices. This method features the possibility of storing non-conserved flow patterns (such as baryon number violation in SUSY models). Some examples of integer flow code representation for several events are provided in Ref. [4].

The Flow class is used to keep track of flow patterns within a graph - each pattern is assigned a unique integer code, and this code is attached to each particle through which it passes. Different flow types are assigned different flow indices, i.e. color flow uses index 1 and 2. Methods are provided to return a particle's flow partners. An example is given at the top of the Flow header file.

4.4.2 HepMC::Polarization

Important Public Methods
- **is_defined**: *returns true if the Polarization has been defined*
- **set_undefined**: *set this to undefined, resetting theta and phi to their default values*

Relevant Data Members
- **theta**: θ *angle in radians* $0 \leq \theta \leq \pi$
- **phi**: ϕ *angle in radians* $0 \leq \phi < 2\pi$

Notes and Conventions
- the angles are robust - if you supply an angle outside the range, it is properly translated (i.e. 4π becomes 0)

Polarization is a data member of GenParticle - its use is optional. It stores the (θ, ϕ) polarization information which can be returned as a ThreeVector as well.

4.4.3 HepMC::FourVector

Important Public Methods
- A number of simple vector manipulations are available. Check the reference manual for details.

Relevant Data Members
- **x**: *position x or momentum px*
- **y**: *position y or momentum py*
- **z**: *position z or momentum pz*

- **t**: *time or energy*

GenParticle momentum and GenVertex position are stored as FourVectors. FourVector has a templated constructor that will automatically convert any other vector with x(), y(), z(), and t() access methods to a FourVector. This feature is used when converting from the HEPEVT common block.

4.5 HepMC::IO_BaseClass

IMPORTANT PUBLIC METHODS

- **write_event**: *writes out the specified event to the output strategy*
- **read_next_event**: *reads the next event from the input strategy into memory*
- **operator<<,operator>>**: *overloaded to give the same results as any of the above methods*

IO_BaseClass is the abstract base class defining the interface and syntax for input and output strategies of events and particle data tables.

Several IO strategies are supplied:

- **IO_GenEvent** uses iostreams for input and output thereby providing a form of persistency for the event record. This class handles all information found in a GenEvent object. This class replaces IO_Ascii and reads both formats. Events may be contained within the same file together with an unlimited number of comments. The examples (Section 8) make use of this class.
- **IO_AsciiParticles** writes events in a format similar to Pythia 6 output. This is intended for human readability.
- **IO_HEPEVT** reads and writes events to/from the Fortran HEPEVT common block. It relies on a helper class HEPEVT_Wrapper which is the interface to the common block (which is defined in the header file HEPEVT_Wrapper.h[4]). This IO strategy provides the means for interfacing to Fortran event generators. Other strategies which interface directly to the specific event record of a generator could be easily implemented in this style. An example of using IO_HEPEVT to transfer events from Pythia into HepMC is given in `example_MyPythia.cc` (Section 8).

Note that as of HepMC 2.05 it is possible to read and write events directly with streaming I/O operators instead of using IO_GenEvent.

5 Overview of Iterators

Examples of using the particle/vertex iterators are provided in `example_UsingIterators.cc` (Section 8). Useful examples can also be found in testHepMCIteration.cc.

As of HepMC 2.06.00, iterator range classes and methods are available for improved functionality.

[4]Different conventions exist for the fortran HEPEVT common block. 4 or 8-byte floating point numbers may be used, and the number of entries is often taken as 4000 or 10000. To account for all possibilities the precision (float or double) and number of entries can be set for the wrapper at run time,

i.e. HEPEVT_Wrapper::set_max_number_entries(4000);
HEPEVT_Wrapper::set_sizeof_real(8); .

To interface properly to HEPEVT and avoid nonsensical results, it is essential to get these definitions right *for your application*. See example_MyPythia.cc (Section 8) for an example.

5.1 HepMC::GenEvent::vertex_iterator

GenEvent::vertex_iterator inherits from std::iterator¡std::forward_iterator_tag,...¿. It walks through all vertices in the event exactly once. It is robust and fast, and provides the best way to loop over all vertices in the event. For each event, vertices_begin() and vertices_end() define the beginning and one-past-the-end of the particle iterator respectively.

GenEventVertexRange(GenEvent&) is a wrapper for GenEvent::vertex_iterator which provides begin() and end() for ease of use with external fuctions, such as foreach. GenEvent::vertex_range() also provides this functionality.

5.2 HepMC::GenEvent::vertex_const_iterator

A constant version of HepMC::GenEvent::vertex_iterator, otherwise identical.

ConstGenEventVertexRange is the constant version of GenEventVertexRange.

5.3 HepMC::GenEvent::particle_iterator

GenEvent::particle_iterator inherits from std::iterator¡std::forward_iterator_tag,...¿. It walks through all particles in the event exactly once. It is robust and fast, and provides the best way to loop over all particles in the event. For each event, particles_begin() and particles_end() define the beginning and one-past-the-end of the particle iterator respectively.

GenEventParticleRange(GenEvent&) is a wrapper for GenEvent::particle_iterator which provides begin() and end() for ease of use with external fuctions, such as foreach. GenEvent::particle_range() also provides this functionality.

5.4 HepMC::GenEvent::particle_const_iterator

A constant version of HepMC::GenEvent::particle_iterator, otherwise identical.

ConstGenEventParticleRange is the constant version of GenEventParticleRange.

5.5 HepMC::GenVertex::vertex_iterator

NOTES AND CONVENTIONS

- the iterator range must be specified to instantiate - choices are: parents, children, family, ancestors, descendants, and relatives
- note: iterating over all ancestors and all descendents is *not* necessarily equivalent to all relatives - this is consitent with the range definitions

GenVertex::vertex_iterator differs from GenEvent::vertex_iterator in that it has both a starting point and a range. The starting point is the vertex - called the root - from which the iterator was instantiated, and the range is defined relative to this point. The possible ranges are defined by an enumeration called HepMC::IteratorRange and the possibilities are:

- **parents**: *walks over all vertices connected to the root via incoming particles*
- **children**: *walks over all vertices connected to the root via outgoing particles*
- **family**: *walks over all vertices connected to the root via incoming or outgoing particles*
- **ancestors**: *walks over all vertices connected to the root via any number of incoming particle edges - i.e. returns the parents, grandparents, great-grandparents, . . .*

- **descendants**: *walks over all vertices connected to the root via any number of outgoing particle edges - i.e. returns the children, grandchildren, great-grandchildren, ...*
- **relatives**: *walks over all vertices belonging to the same particle/vertex graph structure as the root*

The iterator algorithm traverses the graph by converting it to a tree (by "chopping" the edges at the point where a closed cycle connects to an already visited vertex) and returning the vertices in post order. The iterator requires more logic than the GenEvent::vertex_iterator and thus access time is slower (the required to return one vertex goes like $\log n$ where n is the number of vertices already returned by the iterator).

GenVertex::vertex_iterator allows the user to step into a specific part of a particle/vertex graph and obtain targetted information about it.

5.6 HepMC::GenVertex::particle_iterator

Notes and Conventions

- the iterator range must be specified to instantiate - choices are: parents, children, family, ancestors, descendants, and relatives

GenVertex::particle_iterator behaves exactly like GenVertex::vertex_iterator, with the exception that it returns particles rather than vertices. As a particle defines an edge or line (rather than a point) in the particle/vertex graph, it is intuitive to define the particle_iterator relative to a vertex (point in the graph) - thus the starting point (root) is still a vertex, and the range is defined relative to this root. The extension to particles can be made by using the particle's production or end vertex as the root. Possible ranges are defined by an enumeration called HepMC::IteratorRange. The possibilities are:

- **parents**: *walks over all particles incoming to the root*
- **children**: *walks over all particles outgoing from the root*
- **family**: *walks over all particles incoming or outgoing from the root*
- **ancestors**: *walks over all incoming particles or particles incoming to ancestor vertices of the root - i.e. returns the parents, grandparents, great-grandparents, ...*
- **descendants**: *walks over all outgoing particles or particles outgoing to descendant vertices of the root - i.e. returns the children, grandchildren, great-grandchildren, ...*
- **relatives**: *walks over all particles belonging to the same particle/vertex graph structure as the root*

The class is composed of a GenVertex::vertex_iterator - and the same considerations apply.

GenVertexParticleRange(GenVertex&, IteratorRange) is a wrapper for GenVertex::particle_iterator which provides begin() and end() for ease of use with external fuctions, such as foreach. If the HepMC::IteratorRange is unspecified, it defaults to "relatives". GenVertex::particles(IteratorRange) also provides this functionality.

Also provided, are GenParticleProductionRange(GenParticle&, IteratorRange) and GenParticleEndRange(GenParticle&, IteratorRange), which iterate over the GenParticle's production or end vertex. If the vertex is undefined, begin() and end() will throw a std::range_error. If the HepMC::IteratorRange is unspecified, it defaults to "relatives". GenVertex::particles_in(GenParticle&, IteratorRange), GenVertex::particles_out(GenParticle&, IteratorRange), GenParticle::particles_in(IteratorRange), and GenParticle::particles_out(IteratorRange) also provide this functionality.

6 Ascii Output

Ascii output uses begin and end keys to denote blocks of events. The HepMC version is written immediately before the begin key, but is not part of the event block.

Within a block of events, each line of information begins with a single character key denoting the information found on the line.

General GenEvent information is followed by a list of vertices and associated particles. The count of vertices is expected to match the number of vertices specified in the general event information. Each vertex line specifies how many particle lines are associated with the vertex.

As of HepMC 2.05, it is possible to read and write events directly with the streaming I/O operators >> and << instead of using IO_GenEvent. However, the HepMC event block header and footer will not be written automatically if this method is used. The user must call write_HepMC_IO_block_begin and write_HepMC_IO_block_end explicitly. IO_GenEvent uses these operators internally.

We describe the Ascii output here, which persists all information contained in a HepMC GenEvent.

6.1 Basic IO_GenEvent Structure

<u>Block Keys</u>

- **begin event block**: *HepMC::IO_GenEvent-START_EVENT_LISTING*
- **end event block**: *HepMC::IO_GenEvent-END_EVENT_LISTING*

<u>Line Keys</u>

- **E**: *general GenEvent information*
- **N**: *named weights*
- **U**: *momentum and position units*
- **C**: *GenCrossSection information: This line will appear ONLY if GenCrossSection is defined.*
- **H**: *HeavyIon information: This line will contain zeros if there is no associated HeavyIon object.*
- **F**: *PdfInfo information: This line will contain zeros if there is no associated PdfInfo object.*
- **V**: *GenVertex information*
- **P**: *GenParticle information*

Note that the E, N, U, C, H, and F lines contain event header information. There will be one and only one of these lines per event. Other lines with keys not specified above may be interspersed with the header information. The N, U, C, H, and F lines must lie between the E line and the first V line for each event. N, U, C, H, and F lines can appear in any order. The header information is followed immediately by a V (vertex) line. Each vertex line is immediately followed by the P (particle) lines for particles associated with that vertex. For purposes of IO, particles are associated with a vertex if they are in the list of outgoing particles. In addition, if an incoming particle is not an outgoing particle of some other vertex, then it is classified as an "orphan" incoming particle and associated with the vertex IO. In this way, each particle appears only once in the Ascii listing. An example of the basic format written to a file is shown in Figure 4. See Figure 6 for an example of streaming ouput and Figure 5 for an example with GenCrossSection information.

```
HepMC::Version 2.06.00
HepMC::IO_GenEvent-START_EVENT_LISTING
E 9 51 -1.0000000000000000e+00 -1.0000000000000000e+00 -1.0000000000000000e+00 20 0 309 1 2 0 4 2.3000000000000001e-01 3.4000000000000002e-01 1.1000000000000000e-01 6.500
N 4 "0" "1" "2" "3"
U GEV MM
H 23 11 12 15 3 5 0 0 0 1.4499999582767487e-02 0 0 0
F 2 3 3.5000000000000003e-01 6.4999999999999991e-01 8.4499999999999993e+00 2.4499999779912355e+03 4.5499999591265787e+03 230 230
V -1 0 0 0 0 0 1 3 0
P 1 2212 0 0 6.9999999371178146e+03 7.0000000000000000e+03 9.3827000000000005e-01 3 0 0 -1 0
P 3 21 -9.5802904850995474e-01 3.4892974578914365e-01 1.5677975928920182e+01 1.5711094833049845e+01 0 3 0 0 -3 0
P 12 2101 2.5787537037233477e-01 -1.1110299643709216e-01 1.2403958218239170e+03 1.2403959888942973e+03 5.7933000000000001e-01 2 0 0 -9 0
P 25 2 7.0015367813761997e-01 -2.3782674935205150e-01 2.3333682308044050e+00 2.4698753078332274e+00 3.3000000000000002e-01 2 0 0 -15 0
V -2 0 0 0 0 0 1 2 0
P 2 2212 0 0 -6.9999999371178146e+03 7.0000000000000000e+03 9.3827000000000005e-01 3 0 0 -2 0
P 4 1 2.7745239600449745e-01 -1.8469236508822412e-01 -1.2668437617555701e+03 1.2668438056011901e+03 0 3 0 0 -4 0
P 116 2203 -2.7745239600449745e-01 1.8469236508822412e-01 -1.8910900158159372e+03 1.8910902024916190e+03 7.7132999999999996e-01 2 0 0 -15 0
```

Figure 4: Example of the format written to a file. Only the first few lines are shown. Notice that this event has no GenCrossSection information.

```
HepMC::Version 2.06.00
HepMC::IO_GenEvent-START_EVENT_LISTING
E 1 65 -1.0000000000000000e+00 -1.0000000000000000e+00 -1.0000000000000000e+00 20 0 357 1 2 0 3 3.4560000000000002e-01 9.8595999999999995e-01 9.8563000000000001e-01
N 3 "0" "second weight name" "weightName"
U GEV MM
C 3.3260000000000000e-03 1.0000000000000000e-04
V -1 0 0 0 0 0 1 3 0
P 1 2212 0 0 6.9999999371178146e+03 7.0000000000000000e+03 9.3827000000000005e-01 3 0 0 -1 0
P 3 -1 7.2521029125687850e-02 4.0916270130125820e-01 2.3327360517627892e+02 2.3327397528518750e+02 0 3 0 0 -3 0
P 11 2214 -1.1686660480579378e-01 -8.5929732056936881e-03 6.3364496390829606e+02 6.3364619182056367e+02 1.2419302372801875e+00 2 0 0 -25 0
P 91 1 4.4345575680105935e-02 -4.0056972809556451e-01 6.4508531915990204e+02 6.4508552945967551e+02 3.3000000000000002e-01 2 0 0 -26 0
V -2 0 0 0 0 0 1 2 0
P 2 2212 0 0 -6.9999999371178146e+03 7.0000000000000000e+03 9.3827000000000005e-01 3 0 0 -2 0
P 4 2 7.8470425410496383e-02 3.2223590540096692e-01 -1.4955117837986219e+01 1.4958794842314219e+01 0 3 0 0 -4 0
P 95 2103 -7.8470425410496383e-02 -3.2223590540096692e-01 -3.0233822815549665e+03 3.0233823981369064e+03 7.7132999999999996e-01 2 0 0 -28 0
```

Figure 5: Example of the format written to a file when GenCrossSection and named weights are used. Only the first few lines are shown. Notice that the HeavyIon line was not written because it contained no information.

```
E 1 65 -1.0000000000000000e+00 -1.0000000000000000e+00 -1.0000000000000000e+00 20 0 357 1 2 0 0
U GEV MM
F 2 3 3.5000000000000003e-01 6.4999999999999991e-01 8.4499999999999993e+00 2.4499999779912355e+03 4.5499999591265787e+03 230 230
V -1 0 0 0 0 0 1 3 0
P 1 2212 0 0 6.9999999371178146e+03 7.0000000000000000e+03 9.3827000000000005e-01 3 0 0 -1 0
P 3 -1 7.2521029125687850e-02 4.0916270130125820e-01 2.3327360517627892e+02 2.3327397528518750e+02 0 3 0 0 -3 0
P 11 2214 -1.1686660480579378e-01 -8.5929732056936881e-03 6.3364496390829606e+02 6.3364619182056367e+02 1.2419302372801875e+00 2 0 0 -25 0
P 91 1 4.4345575680105935e-02 -4.0056972809556451e-01 6.4508531915990204e+02 6.4508552945967551e+02 3.3000000000000002e-01 2 0 0 -26 0
V -2 0 0 0 0 0 1 2 0
P 2 2212 0 0 -6.9999999371178146e+03 7.0000000000000000e+03 9.3827000000000005e-01 3 0 0 -2 0
P 4 2 7.8470425410496383e-02 3.2223590540096692e-01 -1.4955117837986219e+01 1.4958794842314219e+01 0 3 0 0 -4 0
P 95 2103 -7.8470425410496383e-02 -3.2223590540096692e-01 -3.0233822815549665e+03 3.0233823981369064e+03 7.7132999999999996e-01 2 0 0 -28 0
V -3 0 0 0 0 0 0 1 0
P 5 -1 4.5914398456298945e-02 2.5904843777716324e-01 1.4768981337590043e+02 1.4769004769866316e+02 0 3 0 0 -5 0
```

Figure 6: Example of the streaming output format. Only the first few lines are shown. Notice that the streaming output does not automatically write the IO_GenEvent begin and end block lines. The user may optionally choose to add those ”headers” to the output stream.

6.2 General Event Information

E - general GenEvent information

- **int**: *event number*
- **int**: *number of multi paricle interactions*
- **double**: *event scale*
- **double**: *alpha QCD*
- **double**: *alpha QED*
- **int**: *signal process id*
- **int**: *barcode for signal process vertex*
- **int**: *number of vertices in this event*
- **int**: *barcode for beam particle 1*
- **int**: *barcode for beam particle 2*
- **int**: *number of entries in random state list (may be zero)*
- **long**: *optional list of random state integers*
- **int**: *number of entries in weight list (may be zero)*
- **double**: *optional list of weights*

N - weight names

- **int**: *number of entries in weight name list. Note that the number of entries here and on the E line are required to be exactly the same and in the same order.*
- **std::string**: *list of weight names enclosed in quotes*

U - momentum and position units

- **std::string**: *momentum units (MEV or GEV)*
- **std::string**: *length units (MM or CM)*

C - GenCrossSection information

- **double**: *cross section in pb*
- **double**: *error associated with this cross section in pb*

H - HeavyIon information

- **int**: *Number of hard scatterings*
- **int**: *Number of projectile participants*
- **int**: *Number of target participants*
- **int**: *Number of NN (nucleon-nucleon) collisions*
- **int**: *Number of spectator neutrons*
- **int**: *Number of spectator protons*
- **int**: *Number of N-Nwounded collisions*
- **int**: *Number of Nwounded-N collisons*
- **int**: *Number of Nwounded-Nwounded collisions*
- **float**: *Impact Parameter(fm) of collision*
- **float**: *Azimuthal angle of event plane*
- **float**: *eccentricity of participating nucleons in the transverse plane*
- **float**: *nucleon-nucleon inelastic cross-section*

F - PdfInfo information

- **int**: *flavour code of first parton*
- **int**: *flavour code of second parton*
- **double**: *fraction of beam momentum carried by first parton*
- **double**: *fraction of beam momentum carried by second parton*
- **double**: *Q-scale used in evaluation of PDF's (in GeV)*
- **double**: *x*f(x) for id1, x1, Q*
- **double**: *x*f(x) for id2, x2, Q*
- **int**: *LHAPDF set id of first parton (zero by default)*
- **int**: *LHAPDF set id of second parton (zero by default)*

6.3 Vertices and Particles

<u>V - GenVertex information</u>

- **int**: *barcode*
- **int**: *id*
- **double**: *x*
- **double**: *y*
- **double**: *z*
- **double**: *ctau*
- **int**: *number of "orphan" incoming particles*
- **int**: *number of outgoing particles*
- **int**: *number of entries in weight list (may be zero)*
- **double**: *optional list of weights*

<u>P - GenParticle information</u>

- **int**: *barcode*
- **int**: *PDG id*
- **double**: *px*
- **double**: *py*
- **double**: *pz*
- **double**: *energy*
- **double**: *generated mass*
- **int**: *status code*
- **double**: *Polarization theta*
- **double**: *Polarization phi*
- **int**: *barcode for vertex that has this particle as an incoming particle*
- **int**: *number of entries in flow list (may be zero)*
- **int, int**: *optional code_index and code for each entry in the flow list*

7 Building HepMC

Source code, binary and source code tarballs, bug tracking, etc. are all available from the HepMC web pages [16] at https://savannah.cern.ch/projects/hepmc/.

Source code tarballs are on the download page: http://lcgapp.cern.ch/project/simu/HepMC/download/. Binary downloads are available for some releases. The following recipe is a guideline and should be modified according to taste.

- **download source code tarball**:
- **mkdir cleanDIR**: *Make a new directory to work in.*
- **cd cleanDIR**:
- **tar -xzf HepMCtarball**: *Unwind the tarball you downloaded.*
- **mkdir build install**: *Define directories for building and installation.*
- **cd build**: *This is your real working directory.*
- **../../HepMC-release/configure –prefix=../install –with-momentum=XX –with-length=YY**: *–prefix tells the tools where to install the library and headers. The default install location is /usr/local.*
- **make**: *Compile HepMC.*
- **make check**: *Run the tests. This is optional but strongly recommended.*
- **make install**: *Install everything in your specified directory.*

8 Examples

Examples are provided in the examples directory of the package and are installed in the installation directory under examples/HepMC. Tests, found in the test directory of the package, also provide useful examples. The tests are not installed. Most examples use IO_GenEvent. However, example_PythiaStreamIO.cc illustrates explicit use of the streaming I/O operators.

- **Using the HepMC vertex and particle iterators:** example_UsingIterators.cc
- **Using HepMC with Pythia (Fortran):** example_MyPythia.cc and example_MyPythiaOnlyToHepMC.cc
- **An Event Selection with Pythia Events:** example_MyPythia.cc
- **Event Selection and Ascii IO** example_EventSelection.cc
- **Using HepMC with Herwig:** example_MyHerwig.cc
- **Write an event file and then read it:** example_MyPythia.cc
- **Write an event file and then read it with the streaming I/O operators:** example_PythiaStreamIO.cc
- **Building an Event from Scratch in the HepMC Framework:** example_BuildEventFromScratch.cc
- **Verify that copying generated events behaves as expected:** testHerwigCopies.cc and testPythiaCopies.cc
- **Using the iterator range classes and methods:** HepMC/test/testHepMCIteration.cc

I/O examples are shown in Figures 7, 9. 8, and 10.

Figure 7: Reading HepMC events from a file.

```
// specify an input file
HepMC::IO_GenEvent ascii_in("example.dat",std::ios::in);
// get the first event
HepMC::GenEvent* evt =  ascii_in.read_next_event();
// loop until we run out of events
while ( evt ) {
    // analyze the event
    ...
    // delete the created event from memory
    delete evt;
    // read the next event
    ascii_in >> evt;
}
```

Figure 8: Reading HepMC events from an input stream.

```
// specify an input stream
std::ifstream is( "example_PythiaStreamIO_write.dat" );
// create an empty event
HepMC::GenEvent evt;
// loop over the input stream
while ( is ) {
    // read the event
    evt.read( is );
    // make sure we have a valid event
    if( evt.is_valid() ) {
        // analyze the event
        ...
    }
}
```

Figure 9: Convert events from the HEPEVT common block to HepMC format and write them to a file.

```
// common block conversion methods
HepMC::IO_HEPEVT hepevtio;
// specify an output file
HepMC::IO_GenEvent ascii_out("example.dat",std::ios::out);
for ( int i = 1; i <= maxEvents; i++ ) {
    // convert an event
    HepMC::GenEvent* evt = hepevtio.read_next_event();
    // analyze the event
    ...
    // write the HepMC event
    ascii_out << evt;
    // delete the created event from memory
    delete evt;
}
```

Figure 10: Convert events from the HEPEVT common block to HepMC format and write them to streaming output.

```
// common block conversion methods
HepMC::IO_HEPEVT hepevtio;
// specify an output stream
std::ofstream os( "example_PythiaStreamIO_write.dat" );
for ( int i = 1; i <= maxEvents; i++ ) {
    // generate the event with Pythia, Herwig, etc.
    ...
    // convert an event
    HepMC::GenEvent* evt = hepevtio.read_next_event();
    // analyze the event
    ...
    // write the HepMC event
    evt->write(os);
    // delete the created event from memory
    delete evt;
}
```

9 Deprecated Classes

Two major classes have been deprecated: IO_Ascii and ParticleData. IO_Ascii was deprecated in 2.02.00 and was removed as of HepMC 2.05.00. IO_Ascii has been replaced by IO_GenEvent, which uses iostreams instead of files.

The ParticleData classes, deprecated since HepMC 2.02.00, had become outmoded and would need a lot of work. The ParticleData classes were removed as of HepMC 2.06.00. Instead, we recommend using packages already developed for this purpose, such as HepPDT [17].

IO_ExtendedAscii, introduced in 1.28.00, was deprecated in 2.02.00 and removed as of HepMC 2.04.00.

IO_GenEvent and the streaming input operator can read old files written by either IO_Ascii or IO_ExtendedAscii.

10 Acknowlegements

We would like to acknowlege useful suggestions, consultations, and comments from: Ian Hinchliffe, Pere Mato, H.T. Phillips, Anders Ryd, Maria Smizanska, and Brian Webber. R.D. Schaffer and Lassi Tuura provided many useful suggestions on the package architecture. Thanks to Witold Pokorski and Pere Mato for providing the fixes that make HepMC compile and run on Windows with Microsoft Visual C++.

References

[1] M. Dobbs and J.B. Hansen, "The HepMC C++ Monte Carlo Event Record for High Energy Physics", Computer Physics Communications (to be published) [ATL-SOFT-2000-001].

[2] Pythia 8.1 available at http://www.thep.lu.se/ torbjorn/pythiaaux/present.html.

[3] Herwig++ 2.1 available at http://projects.hepforge.org/herwig/.

[4] E. Boos *et al.*, "Generic user process interface for event generators," arXiv:hep-ph/0109068.

[5] S. Protopopescu, "MC++ Interface". Available from http://ox3.phy.bnl.gov/s̃erban/mcpp/index.html.

[6] "A Class Library for High Energy Physics," (CLHEP). Available from http://wwwasd.web.cern.ch/wwwasd/lhc++/clhep/.

[7] Generator Services Subproject information available at http://lcgapp.cern.ch/project/simu/generator/.

[8] Latest HepMC ChangeLog available at http://simu.cvs.cern.ch/cgi-bin/simu.cgi/simu/HepMC/ChangeLog?view=markup.

[9] A.A. Stepanov, M. Lee, "The Standard Template Library," Hewlett-Packard Laboratories Technical Report HPL-94-34, April 1994, revised July 7, 1995. Available from ftp://butler.hpl.hp.com/stl/.

[10] T. Sjostrand *et al.*, "High-energy physics event generation with PYTHIA 6.1," Comput. Phys. Commun. **135**, 238 (2001).

[11] G. Corcella *et al.*, "Herwig 6: an event generator for Hadron Emission Reactions With Interfering Gluons (including supersymmetric processes)" JHEP **0101**, 010 (2001) [hep-ph/0011363]; hep-ph/0210213.

[12] "the Les Houches Accord PDF Interface," (LHAPDF). Available from http://projects.hepforge.org/lhapdf/.

[13] A. Ryd, D. Lange, "The EvtGen package for simulating particle decays," Computing in High Energy Physics, Chicago, Illinois, USA (1998).

[14] W.-M. Yao *et al.*, "Review of particle physics," Journal of Physics **G33**, 1 (2006). Available from http://pdg.lbl.gov/.

[15] L. Garren, "StdHep 5.05 Monte Carlo Standardization at FNAL," Fermilab PM0091. Available from http://cepa.fnal.gov/psm/stdhep/.

[16] "a C++ Event Record for Monte Carlo Generators," (HepMC). Available from https://savannah.cern.ch/projects/hepmc/.

[17] HepPDT is available at http://savannah.cern.ch/projects/heppdt/.

[18] MCnet is available at http://www.montecarlonet.org/.