Gauss

LHCb Simulation Program

User Guide and Reference Manual

Corresponding to Gauss version v18r3

Author :	I.Belyaev, G.Corti, S.Easo, W.Pokorski, F.Ranjard, P.Robbe
Revision :	1.4
Issue :	1
Created :	November 22, 2004
Last modified :	January 11, 2005

Contents

Chapter 1

Introduction

1.1 Purpose and structure of this document

The Gauss program is the LHCb Monte Carlo simulation application based on the Geant4 [1] toolkit as well as various generator libraries (e.g. PYTHIA). It consists of two stages, the event generation and the detector response simulation which can be run independently. It produces Monte Carlo "hits" as its output which can then be processed by the Boole [2] digitization application. It also provides *MC truth* information allowing to access the history of a particle producing specific "hits".

Gauss is built on the Gaudi framework [5] and provides mechanisms for sequencing algorithms within the framework. Algorithms executed in Gauss have access to all the services currently implemented in Gaudi, and to all data in the Gaudi data stores. Gauss uses CMT [3] for code management. This guide assumes some familiarity with CMT and Gaudi.

This document is a user guide and reference manual for Gauss. It should be used in conjunction with documentation available on the web at *http://cern.ch/lhcb-comp/Simulation/*. Chapter 2 describes how to configure and execute the released version of the program and should be useful to anyone wishing to use Gauss. Chapter 3 describes how to modify the program and should be useful for developers of simulation software. Chapter 4 describes the current functionality and is intended as a reference guide.

1.2 Package structure

Releases of the GAUSS software project can be found in the LHCb software release area, in the directory $LHCBRELEASES/GAUSS/GAUSS_vxry where x and y are respectively the major and the minor Gauss version numbers. A given version of the GAUSS project contains all the CMT Gauss packages specific to that version.

The GAUSS project depends on the LHCB core software project (event model classes, detector description), on the GEANT4 project (Geant4 toolkit) and on the GAUDI framework. The specific versions of different project and packages which a given version of GAUSS depends on can be found in the GaussEnv vxry package.

The Gauss application resides in the Sim/Gauss package of the GAUSS project. This package is used to build and execute the application. It has the following directory structure:

- `doc` release notes
- `job` example job for Linux
- `options` default and example job options
- `cmt` CMT `requirements` file
- `<binary>` platform/compiler dependent directories for the different binaries

1.3 Problem reporting and resolution

1.3.1 Getting help

If your problem cannot be resolved by looking at this guide, you could try using the Gauss discussion mailing list, *lhcb-gauss@cern.ch*.

1.3.2 Reporting problems

If you think you have found a bug in Gauss, or if you would like to request a new feature, please use the Savannah problem reporting system: *http://savannah.cern.ch/projects/gauss* .

1.4 Editor's note

This document is a snapshot of the Gauss software at the time of the release of version v18r3. We have made every effort to ensure that the information it contains is correct, but in the event of any discrepancies between this document and information published on the Web [8], the latter should be regarded as correct, since it is maintained between releases and, in the case of code documentation [9], it is automatically generated from the code. We encourage readers to provide feedback about the structure, contents and correctness of this document and of other LHCb software documentation. Please send your comments to *Florence.Ranjard@cern.ch* or *Gloria.Corti@cern.ch*.

Chapter 2

Configuring and executing a standard Gauss job

2.1 Introduction

The released version of Gauss contains the recommended configuration for a standard LHCb simulation job. This chapter describes how to configure and execute a standard job; familiarity is assumed with CMT [3]. Instructions for modifying the functionality of Gauss are given in Chapter 3.

2.2 Supported platforms

Gauss is currently built and supported on the following platforms:

- Linux RedHat 7.3 with gcc 3.2.3 compiler (-O2 and debug versions provided)

2.3 Setting up the environment

Before starting to work with Gauss, you need to set up CMT. When you log in to lxplus[1], the CMTPATH environment variable is normally set to `$HOME/cmtuser`. This directory is automatically created the first time you login as an LHCb user and is populated with a special file `cmt/project.cmt`. If your working directory is something other than `cmtuser`, modify the CMTPATH accordingly and ensure the special file `cmt/project.cmt` is present there. [2] In case this is not done for you your can setup the CMT environment via the command

```
source $LHCBHOME/scripts/CMT.(c)sh
```

You now have to tell CMT which version of Gauss you wish to work with. You do this with the command

```
GaussEnv v18r3
```

[1] These instructions are given for an LHCb account on lxplus at CERN, but should be very similar for any site with CMT installed.

[2] Previous version of Gauss use an older version of CMT with a different CMTPATH and without the need of the cmt/project.cmt file. Refer to the web instructions on how to still use the previous versions of CMT [4] ($\leq v1r6$) if you need to run a previous version of Gauss.

This command appends to the CMTPATH the directories required by CMT to locate in the release area all packages required by Gauss v18r3.

Even if you wish to run a standard job, you will need to makes some modifications, if only to define the number and the type of events to simulate and the output file of your job. To do so, get a copy of the Gauss package into your CMT working directory:

```
cd $HOME/cmtuser
getpack Sim/Gauss v18r3
cd Sim/Gauss/v18r3
```

2.4 Structure of options files

The directory options contains various files with suffix *.opts* to control the configuration of a Gauss job. The default configuration is steered by the top level option file `Gauss.opts`, where the main parameters you may wish to change are listed. Detailed steering of other options are contained in specialized options file and will be refered to later. Some example options for special configurations are also provided.

2.5 Setting number of events

Before running any job, you certainly want to specify how many events you want to simulate. You do this by changing the following line in the top level job options file, i.e. in `Gauss.opts`:

```
ApplicationMgr.EvtMax = 500;
```

2.6 Controlling the random number sequence: i.e. run ID and event ID

The random number seed (the entire random number sequence used by all the generators in Gauss) is uniquely determined for each event by the the Run ID (RunNumber) and by the event ID (the actual EventNumber in the processing). This allows the generation of independent data sets as well as full reproducibility of events.

In `Gauss.opts` you can set the run ID (run number)

```
Gauss.RunNumber = 100;
```

and the number of the first event

```
Gauss.FirstEventNumber = 1;
```

By setting these two numbers you can generate separate data sets or re-run particular events in order to debug them.

2.7 Defining the event type

When running a Gauss job you will need to specify what type of events you want to simulate. The default is min-bias events (type 30000000). In case you wanted to simulate particular signal

events or BBbar inclusive ones, you should add at the end of the job options (for example as last line of Gauss.opts) a line like:

```
#include ''$DECFILESROOT/options/10000000.opts''
```

where the number in front of '.opts' corresponds to the event type you want to simulate (BBbar inclusive in the above example). Predefined options for the various event types that can be generated are located in the special package `DecFiles`. The version of the DecFile package in use in a given version of Gauss is the latest available at the moment of a Gauss release. It could be necessary to use new event types with an existing Gauss release, in which case a new compatible version of DecFiles can be specified when running a job. by setting the $DECFILESROOT environment variable directly or modifying the Gauss requirements files. You can see what kind of event types are available by looking in $LHCBRELEASES/DBASE/Gen/DecFiles/v6r4/options/ directory or on the web *http://robbep.home.cern.ch/robbep/EvtGen/EventTypes.html*

2.8 Defining the output file

Gauss writes an event data output file (`.sim`) by default. The options to control this output are show in the listing below:

```
1:  ApplicationMgr.OutStream += ''GaussTape.opts'';
2:  GaussTape.Output = ''DATAFILE='PFN:GaussPool_30000000.sim'
                         TYP='POOL_ROOTTREE' OPT='REC' '';
```

To switch off writing of the output file comment out line 1. The name of the output file is specified by modifying line 2. In the example listed will be written in the directory you are running Gauss from.

The content of the `.sim` file is defined in the file `GaussTape.opts`. It should not normally be necessary to modify this content, but you may do so to e.g. write a reduced dataset or additional RICH information for special studies.

2.8.1 Persistent technology

This version of Gauss supports only the common LCG format ("POOL") persistency technology. The last version supporting also the Gaudi "home-grown" technology (`GaudiPoolDB`) was v16r0. The underlying storage media use is ROOT. The technology "TYPE" must be specified in the output file definition: Gauss requires you to use "POOL_ROOTTREE" necessary for writing files.

One feature of POOL is that internal references between files are made using logical file names (LNF). The mapping between LNFs and physical file names (PFN) is done by one or more file catalogues. A Gauss job will not need any input file catalogues but will write an output one. The name of the file catalogue to write [3] is specified via the following option:

```
PoolDbCacheSvc.Catalog =  "xmlcatalog_file:NewCatalog.xml" ;
```

if the line above is missing or commented a catalogue with the name `test_catalog.xml` will be created where you are running from. If a catalog of the same name exists it will be updated with the new output file entry. In order to be able to preserve the knowledge of the LFN when reading back the `.sim` file for later processing (e.g. with Boole) you should save the read-write catalog at the end of the job (for example by merging it with a master one if you are not updating an

[3] in general a list of catalog can be provided, the first in the list is read-write, those following are read only, hence in a standard Gauss job only the first one in the list will be relevant

existing one). If you move the output file to a different location(e.g. copy it to a castor directory) you have to update the catalog file with the new location.

2.9 Defining monitoring histograms and ntuples

Monitoring in Gauss is performed in the `Monitor` sequence. In this sequence predefine ntuples and histograms are filled both for the Generator and the Simulation phases of Gauss. In production the monitor sequence is switched off. To turn it on you will have to uncomment the line

```
ApplicationMgr.TopAlg += ''Sequencer/Monitor'';
```

in `Gauss.opts`. A default configuration for monitoring histograms and ntuples is available in `Monitor.opts`.

2.10 Reading a Gauss output file

When running a Gauss application you will most likely want to either look at some distribution filled during the processing (see Section 2.9) or you will want to process a Gauss output file with the digitization application, Boole. In this case you should refer to the Boole User Guide [2] for details. In the special cases where you will only want to look at event data objects contained in the `.sim` file as produced by Gauss an example options file is available using Gauss itself. The example options, `GaussRead.opts`, will run the same monitor phase as the one that can be run during the production of events but reading the `.sim` file. To read a file you should overwrite the behaviour of a "normal" Gauss job that has no events input data. This is done by setting the line:

```
ApplicationMgr.EvtSel = "";
```

2.11 Running the job

The following assumes that you have already configured CMT and Gauss as explained in Section 2.3. The commands shown assume that you have set your default directory (`cd`) to `$HOME/cmtuser/Sim/Gauss/v18r3`.

Running the job involves three steps: building the executable, setting up the environment (using the setup file created by CMT) and executing the job with the appropriate setting chosen in the steering job options file.

2.11.1 Running Gauss on Linux

To run Gauss interactively on Linux, use the following commands:

```
cd cmt
source setup.csh
make
cd job
../rh73_gcc32/Gauss.exe ../options/Gauss.opts >& Gauss.log&
```

you can also use the defined alias `gauss` which corresponds to calling the executable with the default jobs options (`Gauss.opts`)

Chapter 3

Modifying and debugging Gauss

The configuration options described in Chapter 2 are the most basic ones used to setup a typical production job. In this chapter we study how to modify the behavior of Gauss in a non-standard way. What is described here are the more common examples, but the possibilities are endless - if you know what you are doing!

It should be noted that the Gaudi job options parses always takes into account the last definition of a job option. Therefore, it is always possible to redefine the value of an option by including the same options again. Alternatively, you can simply edit the appropriate job option file provided by Gauss (or one of the packages it is using) and modify the entry there.

3.1 Manipulating Gauss.opts

As it was mentioned at the beginning of this document, Gauss consists basically of two independent stages, event generation and detector response simulation. These two stages can be run consecutively within one job (the default configuration) or separately (in order to study the event generators or to run the simulation using events saved in a file). In main job options file `Gauss.opts` you have

```
ApplicationMgr.TopAlg =
{"GaussInitialisation/GaussInit",
"Sequencer/Generator",
"Sequencer/Simulation"};
```

which defines the two stages (GaussInitialisation is a mandatory algorithm used to perform some standard tasks like initializing the random number generator, creating the event header, etc and should not be removed), and

```
#include "$GAUSSOPTS/Generator.opts"
#include "$GAUSSOPTS/Simulation.opts"
```

which makes the job options files controlling those two stages to be included.

In order, for instance, to run the job with only the generation sequence on, you should remove the last entry in the list of '`TopAlg`' and leave

```
ApplicationMgr.TopAlg =
{"GaussInitialisation/GaussInit",
"Sequencer/Generator"};
```

Alternatively, if you wanted to run the simulation using some already 'pre-generated' events you should have

```
ApplicationMgr.TopAlg =
{"GaussInitialisation/GaussInit",
"Sequencer/Simulation"};
```

and in addition you should specify the input file (containing the generated events)

```
ApplicationMgr.EvtSel = "<file_name>" ;
```

3.2 Manipulation Generator.opts

The default way of running a Gauss job is to generate events (with pile-up) using Pythia [10] and EvtGen [11] (used to decay almost all short-lived particles and in particular b-hadrons). It is very likely, however, that you will want to run Gauss with the so called 'single particle gun', i.e. a very simple generator 'shooting' single particles of given type, energy, etc into the detector. In order to do that you have to properly specify the members of the generation sequence. The default is

```
Generator.Members =
{"SignalDecayAlg", "PythiaAlg", "SetDecayAlg", "EvtDecayAlg"};
Generator.Members += {"SmearVertexAlg"};
```

where the first set of algorithms is responsible for calling the appropriate event generator and decay package (for more details see Chapter 4), while `SmearVertexAlg` is simulating the smearing to the primary vertex which is due to the size of the colliding bunches. In order now to run the single particle generation you should have the members of the generation sequence to be

```
Generator.Members = "ParticleGun";
```

and the pile-up switched off (which doesn't make sense when you generate events with the particle gun)

```
ParticleGun.Mode=0;
```

In theory, one can also think about some more 'exotic' setups, like for example running Pythia generation without the smearing of the primary vertex, or without the pile-up (`PythiaAlg.Mode=0;`) or running the particle gun with the smearing (determined by the nature of the LHC beams) of the 'shooting point'. These, however, do not seem to be of practical use.

Apart from specifying which generator you want to run, you might also want to configure it. The next two subsections briefly discuss how to change the options of Pythia and ParticleGun.

3.2.1 Changing Pythia options

The default options for Pythia (hardcoded in PythiaAlg and repeated in the option file PythiaSettings.opts) correspond to the "LHCb tunning" and normally should not be changed (unless you want to perform some advance generators studies). In order to set any of the Pythia flags you can use the following syntax

```
PythiaAlg.UserPythiaCommand =
{"<common_block_name> <variable_name> <index> <value>"};
```

3.2.2 Changing particle gun options

When generating particles with single particle gun you can specify a number of parameters like, the position of the vertex (in mm)

```
ParticleGun.xVertexMin = 0.0;
ParticleGun.xVertexMax = 0.0;
ParticleGun.yVertexMin = 0.0;
ParticleGun.yVertexMax = 0.0;
ParticleGun.zVertexMin = 0.0;
ParticleGun.zVertexMax = 0.0;
```

the particle type

```
ParticleGun.PdgCode = 211;
```

the mode of the particle gun ('0' means that you specify px, py, pz, while '1' means that you specify θ and ϕ)

```
ParticleGun.GunMode=1;
```

and the momentum magnitude (in GeV) and direction (in rad)

```
ParticleGun.MomentumMax = 10.0;
ParticleGun.MomentumMin = 10.0;
ParticleGun.ThetMin = 1.0;
ParticleGun.ThetMax = 1.0;
ParticleGun.PhiMin = 0.0;
ParticleGun.PhiMax = 0.0;
```

3.3 Manipulating Simulation.opts

The Simulation.opts file configures and controls the execution of the Geant4 simulation. Most of the entries there should never be moved (or removed) and are mandatory for the correct execution of the program. There are, however, a few things that you might want to set up according to your needs.

3.3.1 Printing MC truth

In some cases, for the purpose of debugging, it might be useful to print on the screen the contents of the MCParticle and MCVertex containers (MC truth) that are kept (saved in the file) after the event. In order to do so, you should add (uncomment - see the default options) the following entry to the list of the simulation members

```
Simulation.Members += { "PrintEventAlg" };
```

It is important to note here, that the order of the members is important and in particular the members

```
Simulation.Members = { "GiGaDataStoreAlgorithm/GiGaStore",
"GiGaInputStream/Geo", "GiGaInputStream/Kine"};
```

should always be called first.

3.3.2 Running jobs with different (incomplete) geometry

In some rare cases (to study the behavior of one particular detector, for instance) you might want to run jobs with some parts of the geometry removed. In order to do that you can comment out some of the entries from the following (default) list

```
Geo.StreamItems += {"/dd/Structure/LHCb/Pipe"};
Geo.StreamItems += {"/dd/Structure/LHCb/Magnet"};
Geo.StreamItems += {"/dd/Structure/LHCb/Velo"};
Geo.StreamItems += {"/dd/Structure/LHCb/Velo2Rich1"};
Geo.StreamItems += {"/dd/Structure/LHCb/Rich1"};
Geo.StreamItems += {"/dd/Geometry/Rich1/Rich1Surfaces"};
Geo.StreamItems += {"/dd/Geometry/Rich1/RichHPDSurfaces"};
Geo.StreamItems += {"/dd/Structure/LHCb/OT"};
Geo.StreamItems += {"/dd/Structure/LHCb/IT"};
Geo.StreamItems += {"/dd/Structure/LHCb/Rich2"};
Geo.StreamItems += {"/dd/Geometry/Rich2/Rich2Surfaces"};
Geo.StreamItems += {"/dd/Structure/LHCb/Spd"};
Geo.StreamItems += {"/dd/Structure/LHCb/Converter"};
Geo.StreamItems += {"/dd/Structure/LHCb/Prs"};
Geo.StreamItems += {"/dd/Structure/LHCb/Ecal"};
Geo.StreamItems += {"/dd/Structure/LHCb/Hcal"};
Geo.StreamItems += {"/dd/Structure/LHCb/Muon"};
```

Alternatively, you might also want to run jobs with a completely different (test beam setup, for instance) geometry. In such a case the entire list above, should be replaced by your new entries.

3.3.3 Changing production cuts

There are presently three processes in Geant4 which are controlled (the infrared divergence) using the production cuts: ionization, bremsstrahlung, and /*mu*-pair production. The production cuts exist, therefore, for the electrons, positrons and photons. They can be set via the job options as follows

```
GiGa.PhysicsList = "GiGaPhysListModular/ModularPL";
GiGa.ModularPL.CutForElectron = 10000.0 * m;
GiGa.ModularPL.CutForPositron = 5.0 * mm;
GiGa.ModularPL.CutForGamma = 10.0 * mm;
```

It should be noted here that although the cuts for gammas and for positrons can be changed arbitrarily (to study their effect on the physics), the cut for electrons is set to a very large value and should not be changed (without a appropriate change in the digitization algorithms). The reason for that cut to be so high (infinite in practice) is that we do not want to simulate the delta rays. The effect of the delta rays is presently taken into account at the level of digitization and therefore lowering the production cut for the electrons would make the whole chain inconsistent.

3.3.4 Manipulating physics lists

Gauss uses the concept of 'modular physics lists' i.e. it allows loading several 'modules' which implement specific physics processes. Such an approach is very flexible and particularly useful for example in the case of the optical processes which need to be loaded for the RICH simulation. The user in general does not need to manipulate the physics list. The only place where he or she might want to change the settings is the choice of the 'hadronic physics'. At the present there are two available models in Gauss, LHEP and QGSP (see [1]). The first one (default) is a parametrized model which is supposed to be the fastest one, while the second one being more theory-driven, might in some cases reproduce the data better at the price of slightly lower performance. You can chose of the model commenting out/uncommenting the following lines in Simulation.opts

```
GiGa.ModularPL.PhysicsConstructors += {"HadronPhysicsLHEP"};
//GiGa.ModularPL.PhysicsConstructors += {"HadronPhysicsQGSP"};
```

3.3.5 Manipulating 'actions'

The main experiment-specific part of the Geant4 simulation application is (apart from the geometry and the sensitive detectors) the implementation of the so-called 'actions' (stepping actions, tracking action, event actions, etc) determining what is actually happening within the simulation loop. A more detailed discussion on the LHCb-specific implementation of those 'actions' will follow in Chapter 4. Here we will briefly describe how to control the contents of the MC truth that is saved after each event.

The default settings are that the primary particles are stored (this is in fact a mandatory setting, ensuring consistency of the MC truth), the particles 'marked' (by, for instance sensitive detectors when hits were generated) are stored and the entire decay chains generated by EvtGen are stored.

```
GiGa.TrackSeq.PostTrack.StorePrimaries = true;
GiGa.TrackSeq.PostTrack.StoreMarkedTracks = true;
GiGa.TrackSeq.PostTrack.StoreForcedDecays = true;
```

In addition to that, you can store particles according to their energy

```
GiGa.TrackSeq.PostTrack.StoreByOwnEnergy = true;
GiGa.TrackSeq.PostTrack.OwnEnergyThreshold = 10.0 * GeV;
```

to the type of their 'creator process'

```
GiGa.TrackSeq.PostTrack.StoreByOwnProcess = true;
GiGa.TrackSeq.PostTrack.StoredOwnProcesses = {"Decay"};
```

as well as according to the particle type, daughters' type, daughters' energy and daughters' creator process.

3.3.6 Choosing the magnetic field 'stepper'

Geant4 offers a number of 'steppers' that can be used to integrate the equation of motion of a particles in the magnetic field. You can choose the stepper using

```
GiGaGeo.FieldManager = "GiGaFieldMgr/FieldMgr";
GiGaGeo.FieldMgr.Stepper = "ClassicalRK4";
```

you can also set a number of parameters like minimal step, delta intersection or delta one step (see [1]).

Chapter 4

Reference manual for Gauss v18r3

4.1 Event generation

Gauss uses two different packages to generate events, `Pythia` and `EvtGen`. `Pythia` generates p-p interactions and decays particles. It stops when a "stable" particle for `Pythia` is reached in the decay tree (usually a meson or a hadron). `EvtGen` will then take care of the generation of the decay of this particle.

Three generation methods are implemented in Gauss:

forced fragmentation the b quark in the event is forced to hadronize with the correct light quark to give the correct b hadron type of the signal decay mode,

repeated hadronization when an event contains a b quark, it is hadronized several times until the correct b hadron type is found,

plain Pythia full events are generated and rejected until it contains the correct b hadron type.

The algorithms and tools used for the event generation process are located in the `Gen/GeneratorModules` package.

4.1.1 PythiaAlg

The interface to `Pythia` is implemented in the algorithm *PythiaAlg* of the `GeneratorModules` package. This algorithm takes care of the primary p-p interactions generation and can be used to generate minimum bias, $b\bar{b}$ events, signal events or other types of events ($c\bar{c}$, ...). Hadrons which are known to `EvtGen`[1] are declared stable for Pythia so that Pythia will stop the decay chain when it will find such a stable particle. The content of the event is then written in HepMC format in the container `"/Event/Gen/HepMCEvents"` and is available for subsequent algorithms like for example *EvtDecayAlg* which will decay particles produced by `Pythia`.

In case the generation method is the "forced fragmentation", *PythiaAlg* has to know what should be the b hadron that has to be produced. The algorithm reads this information from the container `"/Event/Gen/SignalDecay"` which is filled by the algorithm *SignalDecayAlg* and which contains an HepMCEvent with only the signal b hadron (that is to say the hadron decaying to the mode described by the event type of the job).

For the 2 other generation methods, the signal b hadron is generated at rest by `EvtGen` inside *PythiaAlg*, using the b flavour determined by `Pythia` (this enables to keep the possible pro-

[1] A particle is known to `EvtGen` if it has a decay table declared in one of the decay files (the user decay file or the generic decay file `DECAY.DEC`).

duction asymmetries). The signal b hadron (generated at rest) is then stored in the container `"Event/Gen/SignalDecay"`.

The algorithm *PythiaAlg* takes care of decaying excited B resonances with `EvtGen` so that a **signal** B hadron might come from a B^{**} decay. The same algorithm also ensures that the correct B and $\bar{B}$ proportions are generated correctly.

Finally, *PythiaAlg* decays the signal B hadron tree with `EvtGen` in case the generation method is "plain Pythia" or the "repeated hadronization".

The generation of pile-up events is implemented in this algorithm. When this option is activated, minimum bias events are generated and associated with b events containing the requested signal decay mode to form full events. There exist 2 different ways of generating pile-up events which can be controled via job options.

- Fixed number of pile-up events. The `Mode` number has to be set to 0 and the number of pile-up events is the value given to the `MeanInt` variable:

```
PythiaAlg.Mode = 0 ;
PythiaAlg.MeanInt = 2 ;
```

- Variable number of pile-up events. The number of pile-up events is randomly generated for each event. First, the time t at which the interaction occurs with respect to the re-fill of the beam is generated with an uniform distribution in the interval between 0 and `FillDuration`. Then the number of pile-up events is generated according to a Poisson law with mean value equal to:

$$N = Le^{-\frac{t}{\texttt{BeamDecayTime}}} \frac{\texttt{TotalXSect}}{10 \times \texttt{CrossRate}} \tag{4.1}$$

  where

$$L = \frac{\texttt{Luminosity} \times \texttt{FillDuration}}{\texttt{BeamDecayTime}\left(1 - e^{-\frac{\texttt{FillDuration}}{\texttt{BeamDecayTime}}}\right)} \tag{4.2}$$

  All variables can be modified by job options. The default values are:

```
PythiaAlg.Mode = 1 ;
PythiaAlg.Luminosity = 2.0 ;
PythiaAlg.FillDuration = 7.0 ;
PythiaAlg.BeamDecayTime = 10.0 ;
PythiaAlg.CrossRate = 30.0 ;
PythiaAlg.TotalXSect = 102.4 ;
```

`Pythia` is generally configured interacting with Fortran common blocks. *PythiaAlg* offers the possibility to modify some of these configuration common blocks using job options. The default settings corresponding to the "LHCb tunning" are hard-coded in *PythiaAlg* and are repeated in the option file `PythiaSettings.opts` of the Gauss package.

The way to interact with `Pythia` configuration is to add commands to the `UserPythiaCommand` vector:

```
PythiaAlg.UserPythiaCommand += { "command1" , "command2" } ;
```

where the format of the command is: `<command> <entry> (<arg1> .... <arg4>)`.

The available commands are listed in Table 4.1.

Beam parameters are also defined through *PythiaAlg*. The default parameters are listed here and can be changed in job option files:

```
PythiaAlg.BeamEnergy = 7000.0 ;
PythiaAlg.VerticalXAngle = 0.0 ;
PythiaAlg.HorizontalXAngle = 285.0 ;
PythiaAlg.Emittance = 0.503 ;
PythiaAlg.Beta = 10.0 ;
```

Cuts can be applied at generator level in order to speed up the generation. The available cuts are:

- *Cut on the angle of the signal particle*: the signal particle is requested to be emitted with an angle with respect to the z axis below the value `ThetaMax`. By default, this value is equal to 400 mrad. If the signal particle is emitted backward (with $p_z < 0$), the full interaction is inverted ($p_z \to -p_z$ and $z \to -z$).
- *Cut on the momentum of the signal particle*: the momentum of the signal particle is requested to be larger that `Pmin`. By default, this value is equal to 0.

The cut parameters can be set by job options:

```
PythiaAlg.ThetaMax = 0.4 ;
PythiaAlg.Pmin = 0.0 ;
```

4.1.2 EvtGenTool

The decay package `EvtGen` is interfaced to Gauss by two algorithms *SignalDecayAlg* and *EvtDecayAlg*.

- *SignalDecayAlg* produces a b hadron at rest and generates the decay chain according to the signal decay mode. `Pythia` is then aksed to force the hadronization of the produced b or $\bar{b}$ quark into the b hadron decided at this stage.
- *EvtDecayAlg* decays generically all particles produced by Pythia which are known to `EvtGen` (that is to say which have a decay table defined in the generic EvtGen decay table DECAY.DEC). This algorithm also substitutes to the signal b hadron produced by `Pythia` the decay chain produced by the first algorithm *SignalDecayAlg* in case of forced fragmentation generation method or in *PythiaAlg* in case of plain `Pythia` or repeated hadronization generation method, boosting it to the correct frame.

The interaction with EvtGen and the generation sequence is achieved through a Gaudi tool called *EvtGenTool*. The tool has several properties which can be modified by job options: the generic decay file name, the user decay file name which defines the signal decay chain to generate, the event type number (which follows conventions described in [14]), the PDG codes of the signal particles to generate and the generation method.

```
ToolSvc.EvtGenTool.DecayFile = "$DECFILESROOT/dkfiles/DECAY.DEC" ;
ToolSvc.EvtGenTool.UserDecayFile = "MyDecayFile.dec" ;
ToolSvc.EvtGenTool.EventType = 99999999 ;
ToolSvc.EvtGenTool.BhadronID = { 511 , -511 } ;
ToolSvc.EvtGenTool.ForcedFragmentation = false ;
ToolSvc.EvtGenTool.RepeatedHadronization = true ;
```

These options are already defined for a number of decay modes together with the corresponding signal decay file. For a given event type N, these options are automatically defined when including the option files "$DECFILESROOT/options/N.opts". The available event types can be found in the package `DecFiles` or here [15].

The BhadronID property can be either :

1. empty to generate minimum bias events,
2. a unique particle code number. In this case only signal events of the indicated b hadron type will be generated.
3. two opposite particle code numbers. A mixture of b hadrons a $\bar{b}$ hadrons will be generated, both of them decaying according to the same signal decay file. This is the setting used in already existing option files which are in the `DecFiles` package.
4. several particle codes. Inclusive generic decays of the indicated meson types will be generated.

If one wants to generate inclusive decays of only one particle or two particles which are charge conjugates, one has to specify it in job options through the variable `InclusiveProduction` which is set to false by default :

```
ToolSvc.EvtGenTool.InclusiveProduction = false ;
```

4.1.2.1 SignalDecayAlg

This algorithm is activated only if the generation method is the "forced fragmentation" method. For the 2 other methods, this algorithm is skipped. The aim of this algorithm is to pre-decay the signal b hadron at rest with `EvtGen` in order for `Pythia` to know how to hadronize the b quark (or the $\bar{b}$ quark) in *PythiaAlg*.

If one wants to generate a mixture of B and $\overline{B}$ hadrons, the algorithm generally generates both flavours with 50 % probability. However, if the decay model used to decay the signal b hadron contains non vanishing CP asymmetries, the correct asymmetry will be generated by the algorithm.

A HepMCEvent is then created to contain the result of this first step and is stored in the container `"/Event/Gen/SignalDecay"` which will be read back by *PythiaAlg* and *EvtDecayAlg* but which can also be saved to be used in generator stand alone studies.

4.1.2.2 EvtDecayAlg

This algorithm generates with `EvtGen` the decay of all particles which have not been decayed by `Pythia` (that is to say practically all particles, and not only the signal b hadron). Particles that will be decayed by `EvtGen` in this algorithm are flagged by the intermediate algorithm *SetDecayAlg*.

EvtDecayAlg reads back the container `"/Event/Gen/HepMCEvents"` and update the HepMCEvents that it contains to add the particles generated by `EvtGen`. The resulting HepMCEvent is again stored in the same container `"/Event/Gen/HepMCEvents"` [2]. The algorithm reads also the content of the container `"/Event/Gen/SignalDecay"` created by *PythiaAlg* or *SignalDecayAlg* depending on the generation method and which is filled by the signal b hadron and its decay products, in the b hadron rest frame. The b hadron decay tree is boosted to the laboratory frame and is then substituted in the full event to the corresponding b hadron generated by `Pythia` but which has been left undecayed up to now.

For convenience or to generate "clean events" (events with only the signal decay chain), the signal b hadron decay chain in the laboratory frame can be stored separatly in a specific container `"/Event/Gen/BHadronTree"`. This possibility is disabled by default but can be activated by job options:

[2] This does not follow the rule that an object in the event store cannot be modified without changing the name of the container.

```
EvtDecayAlg.IsolateSignal = true ;
```

4.1.3 Writing user decay files

The user decay file contains information about how to decay the particles of interest. To decay the particle P into $D_1 D_2 D_3$ according to the model MODEL with the branching ratio Br, the syntax is the following:

```
Decay P
 BR    D1   D2   D3    MODEL (<arg1> <arg2> ...)  ;
Enddecay
```

The decay model MODEL can take one or more arguments that are written after the model name. A list of all available decay models can be found in the EvtGen documentation.

The instruction CDecay automatically defines the CP conjugate mode for the particle anti-P:

```
CDecay anti-P
```

The intruction Decay erases all previous definitions of decay modes for the particle P. If the sum of the branching fractions defined between Decay and Enddecay is not equal to 1, EvtGen will automatically scale the branching fractions to have a sum equal to one.

In order to force a particle to decay into a specific decay mode, it is usually convenient to define an alias to this particle. Then the generic decay modes defined for the particle in DECAY.DEC will not be altered.

Aliases are predefined for b hadrons to be able to force the decay of the signal b hadron and only it. Then only one b hadron in the event will decay according to the decay mode defined for the b hadron alias. All other b hadrons in the event will decay according to the generic decay table DECAY.DEC (unless it is redefined on purpose in the user decay file). Signal B aliases are listed in Table 4.2.

These aliases are activated and available only if the property ToolSvc.EvtGenTool.BhadronID is set to the corresponding PDG Id. The aliases for charge conjugate particles are declared as charged conjugates which means for example that the B^0 alias will oscillate to the $\overline{B}^0$ alias.

Aliases for other particles can be defined at the beginning of the user decay file with the keyword "Alias" followed by the alias name and the particle name for which the alias is created. In order to generate a decay mode and the charge conjugate decay mode, one has to specify which is the charge conjugate of the alias with the keyword "ChargeConj".

The following example illustrates how to generate $B^0 \to D^- \pi^+$ and $\overline{B}^0 \to D^+ \pi^-$ with $D^\pm \to K^\mp \pi^\pm \pi^\pm$:

```
# Define an alias for D+
Alias MyD+ D+
# Define an alias for D-
Alias MyD- D-
# Declare that MyD+ is the charge conjugate of MyD-
ChargeConj MyD+ MyD-

# Decay table for B0sig (predefined alias for B0)
Decay B0sig
# Use D- alias to be able to force D- decay
1.000   MyD-  pi+    PHSP;
Enddecay

# Decay table for anti-B0sig
CDecay anti-B0sig

# Decay table for MyD+
Decay MyD+
1.000   K-  pi+   pi+    D_DALITZ;
Enddecay

# Decay table for MyD-
CDecay MyD-

End
```

An option file has to be written together with each user decay file to set up correctly EvtGen to use it. The option file corresponding to the previous example would be:

```
// The Event Type number
ToolSvc.EvtGenTool.EventType = 99999999 ;
// The signal B Id
// To generate a mixture of B0 and B0bar
ToolSvc.EvtGenTool.BhadronID = { 511 , -511 } ;
// or to generate only B0
ToolSvc.EvtGenTool.BhadronID = { 511 } ;
// The name of the user decay file
ToolSvc.EvtGenTool.UserDecayFile = "MyDec.dec" ;
// The generation method (Repeated fragmentation by default)
ToolSvc.EvtGenTool.ForcedFragmentation = false ;
ToolSvc.EvtGenTool.RepeatedHadronization = true ;
```

4.1.4 Particle properties

The properties of the particles (masses, widths, ...) are taken from the Gaudi `ParticlePropertySvc`. Since `EvtGen` identifies particles with their names, the Gaudi `ParticlePropertySvc` data table (`ParticleTable.txt`) contains also the `EvtGen` name of each particle. In decay files, this EvtGenName string has to be used and the program will stop if it finds in a decay table a particle whose `EvtGen` identifying string has not been declared in the `ParticlePropertySvc`. Then to write user decay files, the names of particles to use can be found in this file at the location `$PARAMFILESROOT/data/ParticleTable.txt`.

`EvtGen` uses as input a file containing all particles parameters. To transmit the properties contained in the Gaudi `ParticlePropertySvc` into `EvtGen`, a temporary file is created by *EvtGenTool* and

is read by the EvtGen engine. This temporary file is called tempPdlFile.txt and is deleted by default at the end of the Gauss job. It is possible to keep it for debugging purposes, modifying the job options:

```
ToolSvc.EvtGenTool.KeepTempEvtFile = true ;
```

Masses of particles with non-zero width will be generated according to a Breit Wigner lineshape by EvtGen and decay times of particles with non-zero lifetime will be generated according to a exponential funtion. In order to simplify the generation, it is better to generate only Dirac like mass distributions for particles with negligible widths or only immediate decays for particles with negligible lifetimes. A cut is defined in *EvtGenTool* below which the width of the particle will be considered to be equal to 0 or below which the lifetime of the particle will be considered to be equal to 0. On the other hand, particles with a very large lifetime do not have to be decayed by EvtGen because they will likely interact in the detector matter and will then be treated by the Simulation part of Gauss. A cut is also applied to set to 0 the lifetime of particles above this threshold. These particles must not have a decay table in any of the EvtGen decay file (the user decay file or the global decay file) and will then be considered as stable by EvtGen which will not assign any decay position to them. The cut values are $c\tau_{min} = 0.1\,\mu$m, $c\tau_{max} = 10^{16}$ mm and $\Gamma_{min} = 1.5$ keV.

In order to unify particle properties amongst all generation modules, Pythia masses, lifetimes and widths are updated and also set to the ParticlePropertySvc values. There are exceptions such as quarks, di-quarks, W, Z, Higgs and other Pythia special particles. Their parameters are kept as determined by Pythia by default. It is important to note that the parameters are the one used by EvtGen, that is to say with the same restrictions on width and lifetime. Pythia generates broad resonances according to a non relativistic Breit Wigner form whereas EvtGen uses a relativistic form by default.

4.1.5 B mixing

B^0-$\overline{B}^0$ and B_s^0-$\overline{B}_s^0$ mixing is activated by default in EvtGen by the main decay file DECAY.DEC. It is however possible to desactivate it in the user decay file with the following keywords:

```
# Desactivate B0-B0bar mixing
noIncoherentB0Mixing
# Desactivate B0s-B0sbar mixing
noIncoherentBsMixing
```

It is also possible to change the mixing parameters used by the generator, Δm and $\Delta\Gamma$:

```
# Set B0 mixing parameters
yesIncoherentB0Mixing <DeltaMd (in s^-1)> <DeltaGammad (in s)>
# Set B0s mixing parameters
yesIncoherentBsMixing <DeltaMs (in s^-1)> <DeltaGammas (in s)>
```

Internally, the mixing is seen in EvtGen as a decay of a B^0 with a lifetime t into a $\overline{B}^0$ with 0 lifetime. Then the second $\overline{B}^0$ decays according to the decay table. The description of the mixing is the same in the HepMC event containing the generator information. But at the end of Gauss, this B hadron will be represented in MCParticle format as a B^0 MCParticle with mixing flag set to true.

4.1.6 CP violation

When the program is asked to produce a mixture of b and $\bar{b}$ hadrons, the flavour of the b hadron is chosen in the algorithm *SignalDecayAlg* in case of the forced fragmentation generation method and is then transmitted to Pythia which will force the hadronization of the b (or $\bar{b}$) quark into this particular hadron.

For the other generation methods, the flavour is decided randomly in the algorithm *PythiaAlg* and the event is rejected until the flavour decided in `EvtGen` matches the flavour generated by `Pythia`.

For non CP decay modes, the probability of both flavours is 50 % but for CP decay modes, it may be different from 50 %. The flavour of the b hadron to produce is in this case decided randomly by EvtGen with the correct asymmetry.

By default, no CP violation is generated. The CP violation is activated choosing a decay model which includes CP violation effects in the user decay file. The generic decay do not include CP violation.

4.1.7 Standalone generation

It is possible to use Gauss with only the generation sequence, that is to say without simulating the detector response with Geant4. For this, an option file is available in the Gauss package, GenStandAlone.opts.

More details can be found at the address [15].

4.1.8 HepMC containers and conversion to Geant4 particles

The Gauss event generation sequence produces particles described in HepMC format and stored in 3 different locations:

- `"/Event/Gen/HepMCEvents"`: the full event generated by `Pythia` and `EvtGen`
- `"/Event/Gen/SignalDecay"`: the b hadron decay tree, at rest, generated by `EvtGen`
- `"/Event/Gen/BHadronTree"`: the b hadron decay tree in the laboratory frame, generated by `EvtGen`

All HepMC particles have a status code property. This code is set to different values which have the following meanings:

1 Stable particle generated by `Pythia`

2 Unstable particle generated and decayed by `Pythia` (or a fragmented parton)

3 Documentation particle generated by `Pythia`

777 An intermediate particle generated and decayed by `EvtGen`

888 Particle generated by `Pythia` or `EvtGen` and which must be decayed by `EvtGen`, that is to say the root of a decay tree generated by `EvtGen`

889 The signal particle in the laboratory frame

998 Signal particle (a b or c hadron) generated at rest by `EvtGen`

999 Stable particle generated by `EvtGen`

After the generation sequence, the generated particles are transformed into G4Particles to be processed by the simulation sequence. Not all particles are transformed into G4Particles: for examples partons are not converted. Also particles that are unknown to Geant4 (like B^{**}) are also ignored by the conversion but the information will always be present in the generation containers which are available in output files.

A particle is converted if:

- the status is 1, 2, 888 or 889, and,
- the particle has no production vertex, or,
- the particle is a hadron, a lepton, a nucleus or a photon and has only one mother particle which is not a hadron, neither a lepton, neither a nucleus nor a photon, or the status of the mother particle is 3, or,
- the particle is a hadron, a lepton, a nucleus or a photon, and has several mother particles or no mother particle but a production vertex (which is the case for particle guns).

When a particle is converted, all the decay chain is also converted, independently of the status code or the type of the particle.

The conversion is implemented in the `Sim/GiGaCnv` package.

4.1.9 Debugging and testing tools

There are algorithms available to print informations about what is generated or to fill ntuples with particles properties.

The algorithm *DumpMC* of `Gen/GeneratorModules` prints on the screen the content of the HepMCEvents generated. The location of the containers one wants to inspect have to be provided as a list by job options (the default is indicated):

```
DumpMC.Addresses = "/Event/Gen/HepMCEvents" ;
```

The output level of the algorithm has to be set to `MSG::INFO`.

DumpMCDecay prints on the screen the decay tree of selected particles in a given HepMCEvents container. The selection is based either on the quark content of the particle or on the PDG number of the particle. The container to inspect as well as the selection parameters can be set by job options (the default values are indicated, the arguments can be a list of values):

```
DumpMCDecay.Addresses =  "/Event/Gen/HepMCEvents"  ;
DumpMCDecay.Particles =  0  ;
DumpMCDecay.Quarks =  5  ;
```

The output level of the algorithm has to be set to `MSG::INFO`.

A monitoring algorithm *GeneratorFullMonitor* is provided in the `Sim/GaussMonitor` package. It fills a ntuple with variables of particles in a given HepMCEvents container. The particles stored in the ntuple are the same than the particles converted into G4Particles. The location of the input container can be modified by job options (illustrated here with the default value):

```
GeneratorFullMonitor.Input = "/Event/Gen/HepMCEvents" ;
```

The variables of the ntuple are:

- **NPart** the number of particles in the event,

- **e(i), px(i), py(i), pz(i)** the 4-vector of the particle i,
- **vxProd(i), vyProd(i), vzProd(i), vtProd(i)** the position of the production vertex of the particle i,
- **vxDecay(i), vyDecay(i), vzDecay(i), vtDecay(i)** the position of the decay vertex of the particle i,
- **pdgId(i)** the pdg code of the particle i,
- **nDau(i)** the number of daughter particles of particle i,
- **pdgIdMother(i)** the pdg code of the mother particle of particle i,
- **pdgIdDau1(i)** the pdg code of the first daughter particle of particle i,
- **pdgIdDau2(i)** the pdg code of the second daughter particle of particle i,
- **pdgIdDau3(i)** the pdg code of the third daughter particle of particle i,
- **pdgIdDau4(i)** the pdg code of the fourth daughter particle of particle i,
- **pdgIdDau5(i)** the pdg code of the fifth daughter particle of particle i,
- **pdgIdDau6(i)** the pdg code of the sixth daughter particle of particle i,
- **indexMother(i)** the index of the mother particle of particle i in the particle block,
- **indexInter(i)** the index of the interaction of particle i in the interaction block,
- **NInter** number of pile-up interactions in the event,
- **isBB(j)** 1 if the interaction j contains a b quark *(not yet implemented)*.

It is also possible to interact directly with `Pythia` to obtain useful informations for debugging. For example, to dump to the file `pythia.out` the `Pythia` decay table and the content of the generated event (by `Pythia` only), add the following lines to the option files:

```
PythiaAlg.UserPythiaCommand +=
{ "pyinit output pythia.out" ,
"pyinit pylisti 12" ,
"pyinit pylistf 3" } ;
```

4.1.10 External Libraries

4.1.10.1 Pythia

Version 6.224.2 from the GENSER library is used. This version uses `HEPEVT` common blocks with a size of 10000 particles. Some routines or functions of `Pythia` are overriden to take into account specific aspects of the generation of events in LHCb:

- *pykfdi*: to add modifications needed to perform the forced fragmentation in `Gauss`. The settings for the forced fragmentation are transmitted through the variables `MSTJ(30-34)` and `MSTU(150-152)` of the `PYDAT1` common block which are otherwise unused in `Pythia`. [Redefined in `GeneratorModules`]
- *pyr*: to use the `Gaudi` random number generator instead of the default `Pythia` one. [Redefined in `GeneratorModules`]
- *pyrand*: to remove advisory warnings about maximum violation but which are not important. [Redefined in `GeneratorModules`]

- *lunhep*: function which is used instead of *pyhepc* to convert `Pythia` events into the `HEPEVT` format. This function also returns the number of b quarks in the event. [Defined in `GeneratorModules`]

- *lutran*: to translate particle codes between PDG standard and Pythia standard which do not agree for some particles. This is currently the case for particles with PDG codes 9000111 ($a_0(980)^0$), 9000211 ($a_0(980)^+$), 9010221 ($f_0(980)$), 10221 ($f_0(1370)$), 10111 ($a_0(1450)^0$) and 10211 ($a_0(1450)^+$). [Defined in `GeneratorModules`]

4.1.10.2 Photos

Version 2.07 from the GENSER library is used by the `EvtGen` package. This version uses `HEPEVT` common blocks with a different size, then in order to work with the size used in `Gauss` (10000). Some routines are redefined in `Gauss`:

- *phocin*: to redefine the common block `PHOQED` with a size of 10000. [Redefined in `EvtGen`]

- *phoinf*: to print correct information about `HEPEVT` common block size (10000). [Redefined in `EvtGen`]

- *photos_get* and *photos_set*: to redifine the common blocks `HEPEVT` and `PHOQED` with a size of 10000. [Redefined in `EvtGen`]

- *phoran*: to use the `Gauss` random number generator instead of the default one in `Photos`. [Redefined in `GeneratorModules`]

4.2 GiGa - interface to Geant4

The interfacing of Geant4 to the Gaudi environment is done by GiGa [12]. It not only provides the flow of data between Gaudi and Geant4 objects but also allows the control of the Geant4 event loop through the standard Gaudi job options.

4.3 Detector simulation - general classes

In this section we will concentrate on the contents of the `Sim/GaussTools` packages since it is there that most of the LHCb-specific classes are implemented.

4.3.1 GaussInitialisation

GaussInitialisation is a small algorithm called at the begining of the simulation sequence which plays two roles. It sets the random number seed for the event using the Hash32 algorithm (also used by Boole and Brunel) as well as it generates the EventHeader.

4.3.2 TrCutsRunAction

`TrCutsRunAction` it is a run action which in its `BeginOfRunAction` method attaches user-defined process to the selected particles. These process are:

4.3.2.1 MinEkineCut

`MinEkineCut` is basically the equivalent of the tracking cut in Geant3. It stops tracking the particle which are below specified energy. This process is attached to all the particles except optical photons and can be configure through the `Simulation/simulation.xml` file from `XmlDDDB`.

4.3.2.2 WorldCut

`WorldCut` is a process with stops tracking the particle which go outside the 'world'. This process is attached to all the particles. The limits can be changed through the job options.

4.3.2.3 LoopCuts

`LoopCuts` is a process which stops particles making too many small steps. This process is attached to electrons.

4.3.3 GaussPreTrackAction

This is a mandatory tracking action which must be called before any other tracking actions. It contains only the `PreUserTrackingAction` method and its role is to instanciate `GaussTrajectory` objects.

4.3.4 GaussPostTrackAction

This tracking action must be called after any other tracking action. Its role is to decide whether the given trajectory is to be kept or not. In case it was not kept, the consistency of the MC truth is assured by setting the parent ID of the daughters to the ID of the last saved trajectory.

There is a number of criteria according to which the decision about storing (or not) the trajectory can be based on. In particular, the primary (i.e. not having a mother trajectory) trajectories should alwalys be stored (for consistency reason).

4.3.5 GaussStepAction

This stepping action is responsible for updating the GaussTrackInformation objects according to what has happened during the particular step. It is needed in order to append steps of interest (where hits where created, reflection of optical photons, etc).

4.4 Detector simulation - subdetector specific classes

4.4.1 GaussTracker

The `GaussTracker` package contains 'sensitive detector' implementations as well as the appropriate hit converters for all the tracker-like devices in LHCb. All the three subdetectors (for the trackers (Inner and Outter), for the Velo and for the Muon) generate hits where the following information is stored: the energy deposition, the entry and exit points, the time of flight and the ID of the track that created the given hit (needed to set the appropriate links between hits and MCParticles). In addition to that, the Velo sensitive detector provides the sensor number for the hit, while the Muon sensitive detector provides the ChamberID and the GapID.

In addition to that, the sensitive detector implementations are responsible for updating `GaussTrackInformation` whenever a new hit assigned to the given track has been created.

4.4.2 GaussCalo

The `GaussCalo` package contains the sensitive detector implementations for the LHCb calorimeter system (PRS, SPD, ECAL and HCAL). All subdetectors produce hits containing the following informations:

- the energy deposited by charged particles during a Geant4 step,
- the time when the energy was deposited,
- the cell ID where the energy was deposited,
- a link to the track which generated the hit.

A "forbidden" volume is defined for each sub-detector. When a hit is created, it is associated to the Geant4 track which generated the hit except if the production vertex of this track is inside this volume. In this case, the hit is associated to the first track in the parent history which has a vertex outside this volume. This volume is defined by its minimum z co-ordinate and its maximum z co-ordinate. By default $z_{min} = -1\,\text{km}$ and $z_{max} = 1\,\text{km}$, that is to say all hits inside the calorimeters are associated to the track which generated it.

In order to save time and space, hits are only associated to tracks with vertices outside the calorimeters' volumes. This is achieved redefining in job options the z_{min} and z_{max} parameters:

```
GiGaGeo.Spd.zMin = 12300.  * mm ; GiGaGeo.Spd.zMax = 15000.  * mm ;
GiGaGeo.Prs.zMin = 12300.  * mm ; GiGaGeo.Prs.zMax = 15000.  * mm ;
GiGaGeo.Ecal.zMin = 12300.  * mm ; GiGaGeo.Ecal.zMax = 15000.  * mm ;
GiGaGeo.Hcal.zMin = 12300.  * mm ; GiGaGeo.Hcal.zMax = 15000.  * mm ;
```

Each individual energy deposition in the sensitive material of the sub-detectors is corrected for saturation effects according to the Birk's law with parameters taken from [13]

$$\text{Correction} = \frac{1}{1 + C_1 \frac{dEdx}{\rho} + C_2 \left(\frac{dEdx}{\rho}\right)^2} \tag{4.3}$$

where C_1 and C_2 are coefficients which can be altered via job options, $dEdx$ is the energy deposition in MeV/cm and ρ the density of the scintillator in g/cm^3. The default values [13] are:

```
GiGaGeo.Ecal.BirkC1 = 0.013 * g/MeV/cm2 ;
GiGaGeo.Ecal.BirkC2 = 9.6E-6 * g*g/MeV/MeV/cm2/cm2 ;
```

(and may also be altered the same way for PRS, SPD and HCAL).

Each individual energy deposition is also shared between consecutive 25 ns integration windows, in order to simulate the behaviour of the signal integration by the electronic chain. The energy is shared between 2 consecutive 25 ns time windows for the ECAL and the HCAL and between 6 consecutive windows for the PRS and the SPD.

The fractions of energy in each time bin are stored in input hbook histograms. The histograms give the fraction of energy for each time window as a function of Δt, the time when the energy is deposited with respect to the time when a photon coming from the origin arrives at the z position of the maximum of the shower in the same cell the energy was deposited.

The location of the input timing histograms may be modified by job options:

```
HistogramDataSvc.Input +=
 "GaussCalo DATAFILE='$PARAMFILESROOT/data/gausscalo.hbook' TYP='HBOOK' " ;
```

A specific ECAL correction is also applied to all individual energy depositions to simulate the local non-uniformity of the detector response. The following correction is applied:

$$\begin{aligned}\text{Correction} = A_{local}\left(1-\cos 2\pi\frac{x-x_0}{d}\right)\left(1-\cos 2\pi\frac{y-y_0}{d}\right)+\\ A_{global}(x-x_0+L/2)^2(y-y_0+L/2)^2+\\ A_{reflectionHeight}(e^{-\frac{|x-x_0+L/2|}{A_{reflectionWidth}}}+e^{-\frac{|x-x_0-L/2|}{A_{reflectionWidth}}}+\\ e^{-\frac{|y-y_0+L/2|}{A_{reflectionWidth}}}+e^{-\frac{|y-y_0-L/2|}{A_{reflectionWidth}}})\end{aligned} \tag{4.4}$$

where (x_0, y_0) are the co-ordinates of the center of one cell, L is the size of one cell and d is the distance between fibers. A_{local}, A_{global}, $A_{reflectionHeight}$ and $A_{reflectionWidth}$ are parameters controlling the amplitudes and shapes of the correction. They depend on the area of the ECAL and may be changed by job options. The default values are:

```
GiGaGeo.Ecal.a_local_inner_ecal = 0.  ;
GiGaGeo.Ecal.a_local_middle_ecal = 0.  ;
GiGaGeo.Ecal.a_local_outer_ecal = 0.  ;
GiGaGeo.Ecal.a_global_inner_ecal = 0.0004 ;
GiGaGeo.Ecal.a_global_middle_ecal = 0.002 ;
GiGaGeo.Ecal.a_global_outer_ecal = 0.03 ;
GiGaGeo.Ecal.a_reflection_height = 0.09 ;
GiGaGeo.Ecal.a_reflection_width = 6.  * mm ;
```

4.4.3 GaussRICH

GaussRICH package contains the software which is specific to the simulation of the two RICH detectors in LHCb. It uses GiGa package which interfaces GEANT4 with Gaudi framework.

The geometry of the RICH detectors is encoded XMLDDDB database. This is read by the DetDesk package into the classes of the LHCb specific geometry framework. The GiGa converts them from this framework into the GEANT4 geometry framework.

GaussRICH uses all the standard GEANT4 electromagnetic, hadronic and optical physics processes as enlisted in the 'physics lists' setup by the GaussPhysics package. Some of the optical physics processes are modified in GaussRICH to suit the needs of the RICH simulation and these are renamed to avoid confusion with their corresponding original GEANT4 versions. Couple of RICH specific physics processes are created in GausRICH which are not available in GEANT4. The classes for these are derived from the appropriate GEANT4 base classes.

The silicon detector in each HPD is the sensitive detector which creates 'hits'. The information related to each hit is stored in RICHG4Hit class which are later converted to MCRichHit and MCRichOpticalPhoton and MCRichSegment classes for storage in the output file.

The following sections describe some of the details of these structures indicated above.

4.4.3.1 RICH1 XML geometry files

All the geometry parameters are defined in the files under the Rich1/GeomParam subdirectory of XMLDDDB. The materials are defined in RichMaterials.xml and RichMaterialTabProperty.xml. The main components of the RICH1 geometry are:

- Magnetic Shielding,
- Aerogel , Filter,

- Spherical and Flat Mirrors,
- GasQuartz Window,
- Array of HPDs,
- Exit Window.
- Rich1 Optical Surfaces

The Array of HPDs are inside a volume made of Nitrogen gas. The Aerogel and Mirrors are inside a volume made of C_4F_{10} gas. All the components near the Beam Pipe have the Beam Pipe subtracted out using boolean subtractions.

4.4.3.1.1 Magnetic Shielding The components of the Magnetic Shielding are:

- Box with boolean subtraction of another box, for region around HPDs (Rich1MgsOuter)
- Box for the part on Upstream side nearest to BeamPipe (Rich1MgsUpstr)
- Box for the part on Downstream side nearest to BeamPipe (Rich1MgsDnstr)
- Box for the part on the Left and Right sides (Rich1MgsSide)
- Box for the shelf part near downstream end (Rich1MgsMid) .
- Box for the part at the upstream in the corner (Rich1MgsUpstrCorner).
- Box for the part outside the TT chamber(Rich1MgsDnstrTT)

The part on the top half is labeled H0 and the part on the bottom half is labeled H1. The shielding is made of soft iron.

4.4.3.1.2 HPD The components of each HPD are:

- Spherical Quartz Input Window
- Spherical PhotoCathode
- Cylindrical Kovar Envelope
- Kovar EndCap:
- Silicon Detector

The Array of HPDs is inside a Photo Detector Support Frame. The arrangement of HPDs is described in the corresponding XML files.

4.4.3.1.3 Aerogel In the Aerogel region there are:

- Four Aerogel tiles around the beam pipe.
- Opaque wrap around the lateral sides and upstream side of a tile.
- Filter downstream of Aerogel (optional)

4.4.3.1.4 Exit Window Region The Exit Window has a layer of G10 followed by a layer of PMI material, which is followed by another layer of G10.

4.4.3.1.5 RICH1 surfaces

The Mirror surfaces are defined between the C_4F_{10} and each of the spherical and flat mirror segments. The HPD quartz window surface is between an HPD and its quartz window. The PhotoCathode Surface is between the HPD Quartz Window and HPD PhotoCathode. There is also a surface between an Hpd and its Kovar Envelope.

4.4.3.2 RICH2 XML geometry files

All the geometry parameters are defined in the files under the Rich2/GeomParam subdirectory. The materials used are defined in RichMaterials.xml and RichMaterialTabProperty.xml. The main components of the RICH2 geometry are:

- Magnetic Shielding,
- Spherical and Flat Mirrors,
- GasQuartz Window,
- Array of HPDs,
- Entrance and Exit Window.
- Rich2 Optical Surfaces

The magnetic shielding is made of various boxes and trapezoids. The spherical mirror consists of several mirror segments with hexagonal edges. The flat mirror also consists of several segments which are rectangular boxes. The arrangement of the HPDs in the arrays is documented in the corresponding xml files directly. The entrance window has a layer of Carbonfibre, a layer of PMI (Rohacell51IG) material and another layer Carbon fibre. The exit window has a layer of Aluminum, a layer of PMI (Rohacell51IG) material and another layer of Aluminum. There is an optical surface between the Rich2Mastervolume and each of the spherical and flat mirror segments.

4.4.3.3 Physics Processes

The description of the standard GEANT4 electromagnetic and hadronic processes can be found in the GEANT4 documentation. The ones which are modified for GaussRICH are as follows:

- RichG4Cerenkov Process: For optimizing the CPU time, two different refractive index tables are created in XML. The first is the 'RINDEX' table which has the full wavelength range. The second is the 'CKVRNDX' table which has a limited refractive index range. The latter table is used in RichG4Cerenkov Process for obtaining the refractive index of the medium for Cherenkov photon production. The RINDEX table is used by other optical processes for the optical photons which are created either in the same medium or in a different medium.

  For RICH specific studies, the following information at the time of photon creation are stored in the 'user track information' of the photon. These information are eventually transferred to the corresponding hits for output files. The list of information are:

  - PDG code of the charged track
  - Photon energy at the time of its production
  - Cherenkov Angle Theta and Phi
  - PDG Mass of the charged track
  - Three Momentum of the charged track
  - Pre step and Post step point locations of the charged track

- RichG4OpRayleigh Process : To save CPU time and to avoid infinite looping of photons inside an aerogel tile, a maximum number of allowed steps (default 5000) for an optical photon is implemented. An photon with more steps than the maximum, is killed. This limit can be set through the Rich options file.

  The number of times an optical photon is Rayleigh scattered is stored in the 'usertrack action ' of the optical photon and copied eventually along with the output hit information.

  In the XML files, the Rayleigh scattering lengths for the various wavelengths is calculated from the 'Clarity' parameter and stored in the 'RAYLEIGH' table.

  The constant amount of absorption at large wavelength is converted into an 'absorption length' table in XML files and hence that table is used directly by the G4OpAbsorption process.

- G4OpBoundaryProcess: The quantum efficiency of the HPDs includes the loss of photons at the HPD quartz window. Hence the transmission at the HPD quartzwindow and photocathode are set to be 1.0 to avoid double counting the losses. The HPDMaster uses vacuum as the medium. When a photon goes from the surrounding nitrogen to this vacuum of HPDMaster the transmission is set to 1.0 to avoid unnecessary loss of photons.

  To avoid infinite looping of optical photons due to total internal reflection, the photons undergoing total internal reflection are absorbed in Aerogel, filter and the Gas window of both RICH detectors. This situation can happen when the critical angle on the surface of the rectangular box is smaller than the Cherenkov angle of the photons created in the same box.

The RICH specific physics processes are :

- RichHpdPhotoElectricEffect: The standard G4Photoelectric effect requires one to know the exact material composition of the photocathode to derive the Quantum Efficiency (QE). The RichHpdPhotoElectricEffect uses the QE values from the manufacturer as encoded in the XML table to determine the probability for photoelectron creation. The photoelectrons are created with a direction set by the cross focusing formula and the Point Spread Function (PSF). The particle produced in this process is called 'RichPhotoElectron' which has all the properties of an electron, but has only the transportation and RichHpdSiEnergyLoss processes associated with it. This is to avoid the unnecessary loss of photoelectrons inside an HPD. The various parameters of the HPD are read from XML database into the RichHpdProperties class for using in this process.

- RichHpdSiEnergyLoss: The RichHpdSiEnergyLoss process lets the photoelectrons deposit all their energy inside the silicon. The high energy charged particles that traverse the Silicon deposit the MIP energy in the silicon and continue their transport. Any photoelectron which hit the kovar envelop is killed. The energy stored by this process is eventually copied as the energy of 'hit' in Silicon detector.

4.4.3.4 RICH Hits

The charged particles create the optical photons and photons create the photoelectrons. The photoelectrons which are incident on the HPD silicon detector create the hits. The hits are stored in RichG4Hit class. This class contains location and energy of the hits.

The 'usertrack information' of the photons are converted to the 'usertrack information' of the photoelectrons which in turn are copied to the RichG4Hit class. The full list of information in RichG4Hit class can be found class definition (RichG4Hit.hh). There are four GEANT4 Hit Collection lists. They are, for hits from :

- Top Half of RICH1,
- Bottom Half of RICH1,

- Left Half of RICH2,
- Right half of RICH2.

Some of the information from RichG4Hit class are converted into MCRichHit for standard production runs. The rest of the information are converted into MCRichOpticalPhoton, MCRichSegment classes during special production runs where 'extended' information output is activated. The MCRichHit, MCRichOpticalPhoton and MCRichSegment segment classes are written to the output files according to the options activated.

4.4.3.5 Optimization and Adaptation for Special Studies

In order to optimize the CPU time, the number of photons that are tracked are minimized. For this:

- In RichG4TrackActionPhotOpt, the QE and the maximum reflectivity of the mirrors are taken from the XML to determine the fraction of optical photons to be killed at the PreUserTracking action level of each photon.
- In Rich1G4TrackActionUpstrPhoton, the optical photons which originate upstream of the aerogel are killed at the PreUserTracking action level.
- For the production runs, the aerogel refractive index range in XML for photon production is set to be below the wavelength cut off for the filter.
- The RichG4OptBoundary process has some modifications as mentioned in the previous section on Physics Processes.

For special studies additional ('extended') information is provided in RichG4Hit. The list of additional information is available in RichG4HitClass. Some of these are done in RichG4Cerenkov process and RichG4OpRayleigh Process as indicated in the previous section on Physics Processes. The other information are acquired using 'UserStepActions' in RichG4StepAnalysis3 and RichG4StepAnalysis5. The actual implementations are in the functions in RichG4CherenkovPhotProdTag,RichG4RayleighTag, RichG4AgelExitTag and RichG4MirrorReflPointTag files. The information tagged by RichG4CherenkovPhotProdTag is mentioned above. The RichG4RayleighTag tags the number of times a photon undergoes Rayleigh scattering. The RichG4AgelExitTag tags the location of the photon at the exit plane of Aerogel. The RichG4MirrorReflPointTag tags the reflection points on the spherical and flat mirrors. In addition to these, the location where the photoelectron is originated is tagged in RichHpdPhotoelectricEffect process.

4.4.3.6 Options files

The Rich.opts file activates options specific to GaussRICH. This file has the information needed to activate the various options. From this file one can activate the RichVerbose.opts and RichAnalysis.opts files. If the RichVerbose.opts is activated, the 'extended' information is created and written out to the RichG4Hit class. If the RichAnalysis.opts is activated, some of the built in analysis to count the photoelectronyield and resolutions are activated.

For standard production runs, the Rich.opts is used without the RichVerbose.opts and RichAnalysis.opts.

command	entry	Pythia equivalent
pyinit	pbar	To produce proton anti-proton collisions
	win	Not used
	pylisti	CALL PYLIST(arg1) just after PYINIT
	pylistf	CALL PYLIST(arg1) after all PYEXEC or PYEVNT call
	output	Redirect all Pythia output to file arg1 instead of screen
pysubs	msel	MSEL = arg1
	msub	MSUB(arg1) = arg2
	ckin	CKIN(arg1) = arg2
	kfin	KFIN(arg1,arg2) = arg3
pypars	mstp	MSTP(arg1) = arg2
	msti	MSTI(arg1) = arg2
	parp	PARP(arg1) = arg2
	pari	PARI(arg1) = arg2
pydat1	mstu	MSTU(arg1) = arg2
	mstj	MSTJ(arg1) = arg2
	paru	PARU(arg1) = arg2
	parj	PARJ(arg1) = arg2
pydat2	kchg	KCHG(arg1,arg2) = arg3
	pmas	PMAS(arg1,arg2) = arg3
	parf	PARF(arg1) = arg2
	vckm	VCKM(arg1,arg2) = arg3
pydat3	mdcy	MDCY(PYCOMP(arg1),arg2) = arg3
	mdme	MDME(arg1,arg2) = arg3
	brat	BRAT(arg1) = arg2
	kfdp	KFDP(arg1,arg2) = arg3
pydatr	mrpy	MRPY(arg1) = arg2
	rrpy	RRPY(arg1) = arg2
pymssm	imss	IMSS(arg1) = arg2
	rmss	RMSS(arg1) = arg2
pyint2	iset	ISET(arg1) = arg2
	kfpr	KFPR(arg1,arg2) = arg3
	coef	COEF(arg1,arg2) = arg3
	icol	ICOL(arg1,arg2,arg3) = arg4

Table 4.1: Pythia commands

PDG Id	Particle	EvtGen alias
511	B^0	B0sig
-511	$\overline{B}^0$	anti-B0sig
521	B^+	B+sig
-521	B^-	B-sig
531	B_s^0	B_s0sig
-531	$\overline{B}_s^0$	anti-B_s0sig
541	B_c^+	B_c+sig
-541	B_c^-	B_s-sig
5122	Λ_b	Lambda_b0sig
-5122	$\overline{\Lambda}_b$	anti-Lambda_b0sig
551	η_b	eta_bsig
10553	h_b	h_bsig
5112	Σ_b^-	Sigma_b-sig
-5112	$\overline{\Sigma}_b^+$	anti-Sigma_b+sig
443	J/ψ	J/psisig
421	D^0	D0sig
-421	$\overline{D}^0$	anti-D0sig
411	D^+	D+sig
-411	D^-	D-sig
431	D_s^+	D_s+sig
-431	D_s^-	D_s-sig
4122	Λ_c^+	Lambda_c+sig
-4122	$\overline{\Lambda}_c^-$	anti-Lambda_c-sig

Table 4.2: Predefined EvtGen aliases for signal user decay file

Bibliography

[1] http://geant4.web.cern.ch/geant4

[2] http://cern.ch/lhcb-comp/Digitization/

[3] http://cern.ch/lhcb-comp/Support/CMT/cmt.htm

[4] http://cern.ch/lhcb-comp/

[5] http://proj-gaudi.web.cern.ch/proj-gaudi/

[6] http://cern.ch/lhcb-comp/Frameworks/EventModel/

[7] http://cern.ch/lhcb-comp/Frameworks/DetDesc/default.htm

[8] http://cern.ch/lhcb-comp/Simulation

[9] http://lhcb-release-area.web.cern.ch/LHCb-release-area/GAUSS/doc/html/index.html

[10] http://www.thep.lu.se/tf2/staff/torbjorn/Pythia.html

[11] http://www.slac.stanford.edu/ lange/EvtGen/

[12]

[13] R. L. Graun and D. L. Smith, Nucl. Intrum. Meth. **80**, 239 (1970)

[14] http://lhcb-comp.web.cern.ch/lhcb-comp/event_types_v2.0.pdf

[15] http://lhcb-comp.web.cern.ch/lhcb-comp/Simulation/evtgen.htm