# Book For Application Developers

*Release 11.2*

**Geant4 Collaboration**

**Rev8.0 - December 8th, 2023**

# CONTENTS

**Scope of this manual**

The User's Guide for Application Developers is the first manual the reader should consult when learning about GEANT4 or developing a GEANT4 -based detector simulation program. This manual is designed to:

- introduce the first-time user to the GEANT4 object-oriented detector simulation toolkit,
- provide a description of the available tools and how to use them, and
- supply the practical information required to develop and run simulation applications which may be used in real experiments.

This manual is intended to be an overview of the toolkit, rather than an exhaustive treatment of it. Related physics discussions are not included unless required for the description of a particular tool. Detailed discussions of the physics included in GEANT4 can be found in the Physics Reference Manual. Details of the design and functionality of the GEANT4 classes can be found in the User's Guide for Toolkit Developers.

GEANT4 is a detector simulation toolkit written in the C++ language. The reader is assumed to have a basic knowledge of object-oriented programming using C++. Although GEANT4 is a fairly complicated software system, only a relatively small part of it needs to be understood in order to begin developing detector simulation applications. An understanding of radiation physics and associated processes is beneficial.

# CHAPTER ONE

# INTRODUCTION

## 1.1 How to use this manual

A very basic introduction to GEANT4 is presented in Section *Getting Started with Geant4 - Running a Simple Example*. It is a recipe for writing and running a simple GEANT4 application program. New users of GEANT4 should read this chapter first. It is strongly recommended that this chapter be read in conjunction with a GEANT4 system installed and running on your computer. It is helpful to run the provided examples as they are discussed in the manual. To install the GEANT4 system on your computer, please refer to the Installation Guide for Setting up Geant4 in Your Computing Environment.

Section *Toolkit Fundamentals* discusses general GEANT4 issues such as class categories and the physical units system. It goes on to discuss runs and events, which are the basic units of a simulation.

Section *Detector Definition and Response* describes how to construct a detector from customized materials and geometric shapes, and embed it in electromagnetic fields. It also describes how to make the detector sensitive to particles passing through it and how to store this information.

How particles are propagated through a material is treated in Section *Tracking and Physics*. The GEANT4 "philosophy" of particle tracking is presented along with summaries of the physics processes provided by the toolkit. The definition and implementation of GEANT4 particles is discussed and a list of particle properties is provided.

Section *User Actions* is a description of the "user hooks" by which the simulation code may be customized to perform special tasks.

Section *Control* provides a summary of the commands available to the user to control the execution of the simulation. After Chapter 2, Chapters 6 and 7 are of foremost importance to the new application developer.

The display of detector geometry, tracks and events may be incorporated into a simulation application by using the tools described in Section *Visualization*.

Section *Examples* provides a set of basic, novice, extended and advanced simulation codes which may be compiled and run "as is" from the GEANT4 source code. These examples may be used as educational tools or as base code from which more complex applications are developed.

CHAPTER

TWO

# GETTING STARTED WITH GEANT4 - RUNNING A SIMPLE EXAMPLE

## 2.1 How to Define the main() Program

### 2.1.1 A Sample main() Method

The contents of `main()` will vary according to the needs of a given simulation application and therefore must be supplied by the user. The GEANT4 toolkit does not provide a `main()` method, but a sample is provided here as a guide to the beginning user. Listing 2.1 is the simplest example of `main()` required to build a simulation program.

Listing 2.1: Simplest example of main()

```
#include "DetectorConstruction.hh"
#include "PhysicsList.hh"
#include "ActionInitialization01.hh"

#include "G4RunManagerFactory.hh"
#include "G4UImanager.hh"

int main()
{
  // construct the default run manager
  auto runManager = G4RunManagerFactory::CreateRunManager();

  // set mandatory initialization classes
  runManager->SetUserInitialization(new DetectorConstruction);
  runManager->SetUserInitialization(new PhysicsList);
  runManager->SetUserInitialization(new ActionInitialization);

  // initialize G4 kernel
  runManager->Initialize();

  // get the pointer to the UI manager and set verbosities
  G4UImanager* UI = G4UImanager::GetUIpointer();
  UI->ApplyCommand("/run/verbose 1");
  UI->ApplyCommand("/event/verbose 1");
  UI->ApplyCommand("/tracking/verbose 1");

  // start a run
  int numberOfEvent = 3;
  runManager->BeamOn(numberOfEvent);

  // job termination
  delete runManager;
  return 0;
}
```

The `main()` method is implemented by two toolkit classes, `G4RunManager` and `G4UImanager`, and three classes, `DetectorConstruction`, `PhysicsList` and `ActionInitialization`, which are derived from toolkit classes. Each of these are explained in the following sections.

## 2.1.2 `G4RunManager`

The first thing `main()` must do is create an instance of the `G4RunManager` class. This is the only manager class in the GEANT4 kernel which should be explicitly constructed in the user's `main()`. It controls the flow of the program and manages the event loop(s) within a run. `G4RunManagerFactory::CreateRunManager()` instantiates a `G4RunManager` object whose concrete type is:

- `G4MTRunManager` if Geant4 library was built with multithreading support
- `G4RunManager` otherwise

The concrete type chosen may be overridden at application runtime without recompilation by setting the environment variable `G4RUN_MANAGER_TYPE`, whose value can be set to either `Serial`, `MT`, `Tasking` or `TBB`. For Geant4 version 10.7, options `Tasking` and `TBB` are provided as beta-release. The traditional style of direct instantiation of `G4RunManager` (sequential mode) or `G4MTRunMabager` (multithreaded mode) is also available.

When `G4RunManager` is created, the other major manager classes are also created. They are deleted automatically when `G4RunManager` is deleted. The run manager is also responsible for managing initialization procedures, including methods in the user initialization classes. Through these the run manager must be given all the information necessary to build and run the simulation, including

1. how the detector should be constructed,
2. all the particles and all the physics processes to be simulated,
3. how the primary particle(s) in an event should be produced, and
4. any additional requirements of the simulation.

In the sample `main()` the lines

```
runManager->SetUserInitialization(new DetectorConstruction);
runManager->SetUserInitialization(new PhysicsList);
runManager->SetUserInitialization(new ActionInitialization);
```

create objects which specify the detector geometry, physics processes and primary particle, respectively, and pass their pointers to the run manager. `DetectorConstruction` is an example of a user initialization class which is derived from `G4VUserDetectorConstruction`. This is where the user describes the entire detector setup, including

- its geometry,
- the materials used in its construction,
- a definition of its sensitive regions and
- the readout schemes of the sensitive regions.

Similarly `PhysicsList` is derived from `G4VUserPhysicsList` and requires the user to define

- the particles to be used in the simulation,
- all the physics processes to be simulated.

User can also override the default implementation for

- the range cuts for these particles and

Also `ActionInitialization` is derived from `G4VUserActionInitialization` and requires the user to define

- so-called user action classes (see next section) that are invoked during the simulation,
- which includes one mandatory user action to define the primary particles.

The next instruction

```
runManager->Initialize();
```

performs the detector construction, creates the physics processes, calculates cross sections and otherwise sets up the run. The final run manager method in `main()`

```
int numberOfEvent = 3;
runManager->beamOn(numberOfEvent);
```

begins a run of three sequentially processed events. The `beamOn()` method may be invoked any number of times within `main()` with each invocation representing a separate run. Once a run has begun neither the detector setup nor the physics processes may be changed. They may be changed between runs, however, as described in *Customizing the Run Manager*. More information on `G4RunManager` in general is found in *Run*.

As mentioned above, other manager classes are created when the run manager is created. One of these is the user interface manager, `G4UImanager`. In `main()` a pointer to the interface manager must be obtained

```
G4UImanager* UI = G4UImanager::getUIpointer();
```

in order for the user to issue commands to the program. In the present example the `applyCommand()` method is called three times to direct the program to print out information at the run, event and tracking levels of simulation. A wide range of commands is available which allows the user detailed control of the simulation. A list of these commands can be found in *Built-in Commands*.

## 2.1.3 User Initialization and Action Classes

### User Classes

There are two kinds of user classes, user initialization classes and user action classes. User initialization classes are used during the initialization phase, while user action classes are used during the run. User initialization classes should be directly set to `G4RunManager` through `SetUserInitialization()` method, while user action classes should be defined in `G4VUserActionInitialization` class.

### User Initialization Classes

All three user initialization classes are mandatory. They must be derived from the abstract base classes provided by GEANT4:

- `G4VUserDetectorConstruction`
- `G4VUserPhysicsList`
- `G4VUserActionInitialization`

GEANT4 does not provide default behavior for these classes. `G4RunManager` checks for the existence of these mandatory classes when the `Initialize()` and `BeamOn()` methods are invoked.

As mentioned in the previous section, `G4VUserDetectorConstruction` requires the user to define the detector and `G4VUserPhysicsList` requires the user to define the physics. Detector definition will be discussed in Sections *How to Define a Detector Geometry* and *How to Specify Materials in the Detector*. Physics definition will be discussed in *How to Specify Particles* and *How to Specify Physics Processes*. The user action `G4VUserPrimaryGeneratorAction` requires that the initial event state be defined. Primary event generation will be discussed in *How to Make an Executable Program*.

`G4VUserActionInitialization` should include at least one mandatory user action class `G4VUserPrimaryGeneratorAction`. All user action classes are described in the next section.

Listing 2.2: Simplest example of ActionInitialization

```
#include "ActionInitialization.hh"
#include "PrimaryGeneratorAction.hh"

void ActionInitialization::Build() const
{
  SetUserAction(new PrimaryGeneratorAction);
}
```

### User Action Classes

`G4VUserPrimaryGeneratorAction` is a mandatory class the user has to provide. It creates an instance of a primary particle generator. `PrimaryGeneratorAction` is an example of a user action class which is derived from `G4VUserPrimaryGeneratorAction`. In this class the user must describe the initial state of the primary event. This class has a public virtual method named `GeneratePrimaries()` which will be invoked at the beginning of each event. Details will be given in *How to Generate a Primary Event*. Note that GEANT4 does not provide any default behavior for generating a primary event.

GEANT4 provides additional five user hook classes:

- `G4UserRunAction`
- `G4UserEventAction`
- `G4UserStackingAction`
- `G4UserTrackingAction`
- `G4UserSteppingAction`

These optional user action classes have several virtual methods which allow the specification of additional procedures at all levels of the simulation application. Details of the user initialization and action classes are provided in *User Actions*.

## 2.1.4 `G4UImanager` and UI CommandSubmission

GEANT4 provides a category named **intercoms**. `G4UImanager` is the manager class of this category. Using the functionalities of this category, you can invoke **set** methods of class objects of which you do not know the pointer. In Listing 2.3, the verbosities of various GEANT4 manager classes are set. Detailed mechanism description and usage of **intercoms** will be given in the next chapter, with a list of available commands. Command submission can be done all through the application.

Listing 2.3: An example of main() using interactive terminal.

```
#include "DetectorConstruction.hh"
#include "PhysicsList.hh"
#include "PrimaryGeneratorAction.hh"

#include "G4RunManager.hh"
#include "G4UImanager.hh"

#include "G4UIExecutive.hh"

int main(int argc,char** argv)
{
  // construct the default run manager
  G4RunManager* runManager = new G4RunManager;

  // set mandatory initialization classes
  runManager->SetUserInitialization(new DetectorConstruction);
  runManager->SetUserInitialization(new PhysicsList);
```

(continues on next page)

(continued from previous page)

```
  // set mandatory user action class
  runManager->SetUserAction(new PrimaryGeneratorAction);

  // initialize G4 kernel
  runManager->Initialize();

  // Get the pointer to the User Interface manager
  G4UImanager* UImanager = G4UImanager::GetUIpointer();

  if ( argc == 1 ) {
    // interactive mode : define UI session
    G4UIExecutive* ui = new G4UIExecutive(argc, argv);
    UImanager->ApplyCommand("/control/execute init.mac");
    ui->SessionStart();
    delete ui;
  }
  else {
    // batch mode
    G4String command = "/control/execute ";
    G4String fileName = argv[1];
    UImanager->ApplyCommand(command+fileName);
  }

  // job termination
  delete runManager;
  return 0;
}
```

### 2.1.5 `G4cout`, `G4cerr` and `G4debug`

Although not yet included in the above examples, output streams will be needed. `G4cout` and `G4cerr` are **iostream** objects defined by GEANT4. The usage of these objects is exactly the same as the ordinary *cout* and *cerr*, except that the output streams will be handled by `G4UImanager`. Thus, output strings may be displayed on another window or stored in a file. Manipulation of these output streams will be described in *How to control the output of G4cout/G4cerr*. These objects should be used instead of the ordinary *cout* and *cerr*.

Similarly, `G4debug` may be used for your debug statements. In the Qt GUI lines are highlighted to help you pick out your debug information.

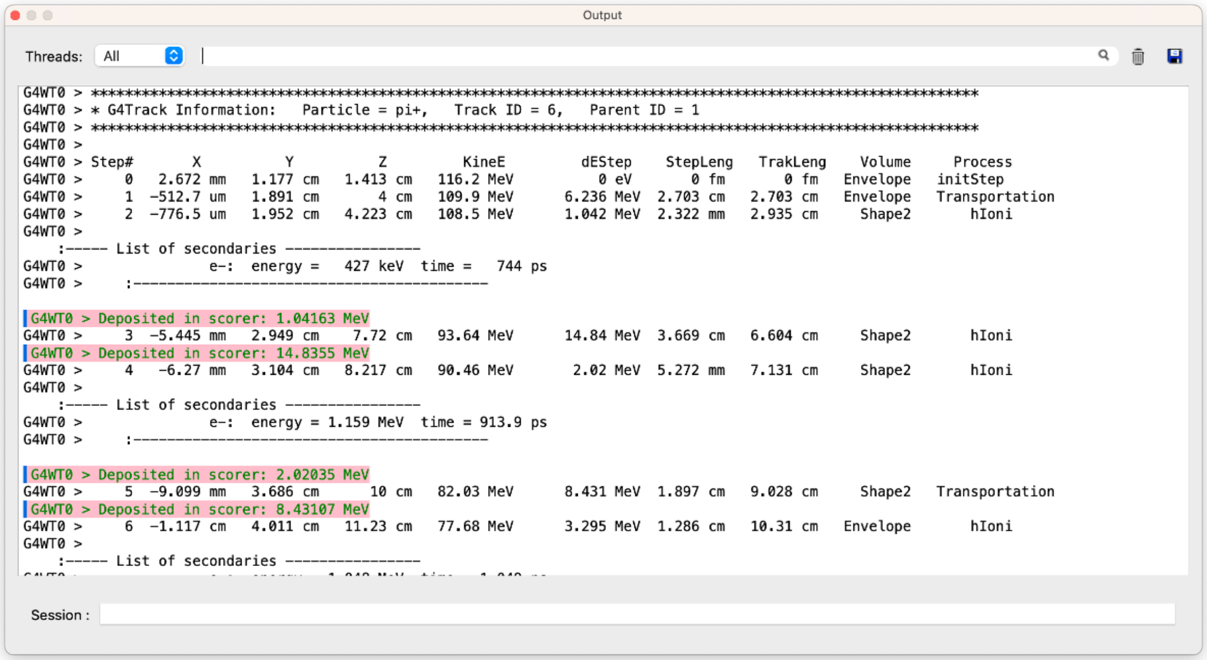
## 2.2 How to Define a Detector Geometry

### 2.2.1 Basic Concepts

A detector geometry in GEANT4 is made of a number of volumes. The largest volume is called the **World** volume. It must contain, with some margin, all other volumes in the detector geometry. The other volumes are created and placed inside previous volumes, included in the World volume. The most simple (and efficient) shape to describe the World is a box.

Each volume is created by describing its shape and its physical characteristics, and then placing it inside a containing volume.

When a volume is placed within another volume, we call the former volume the daughter volume and the latter the mother volume. The coordinate system used to specify where the daughter volume is placed, is the coordinate system of the mother volume.



```
G4WT0 > ******************************************************************************************************
G4WT0 > * G4Track Information:   Particle = pi+,   Track ID = 6,   Parent ID = 1
G4WT0 > ******************************************************************************************************
G4WT0 >
G4WT0 > Step#      X         Y         Z        KineE      dEStep   StepLeng  TrakLeng    Volume     Process
G4WT0 >     0   2.672 mm   1.177 cm   1.413 cm   116.2 MeV        0 eV      0 fm      0 fm   Envelope  initStep
G4WT0 >     1  -512.7 um   1.891 cm      4 cm   109.9 MeV     6.236 MeV  2.703 cm  2.703 cm   Envelope  Transportation
G4WT0 >     2  -776.5 um   1.952 cm   4.223 cm   108.5 MeV     1.042 MeV  2.322 mm  2.935 cm    Shape2      hIoni
G4WT0 >
    :----- List of secondaries ----------------
G4WT0 >              e-:  energy =   427 keV  time =   744 ps
G4WT0 >     :------------------------------------------

G4WT0 > Deposited in scorer: 1.04163 MeV
G4WT0 >     3  -5.445 mm   2.949 cm    7.72 cm   93.64 MeV     14.84 MeV  3.669 cm  6.604 cm    Shape2      hIoni
G4WT0 > Deposited in scorer: 14.8355 MeV
G4WT0 >     4   -6.27 mm   3.104 cm   8.217 cm   90.46 MeV      2.02 MeV  5.272 mm  7.131 cm    Shape2      hIoni
G4WT0 >
    :----- List of secondaries ----------------
G4WT0 >              e-:  energy = 1.159 MeV  time = 913.9 ps
G4WT0 >     :------------------------------------------

G4WT0 > Deposited in scorer: 2.02035 MeV
G4WT0 >     5  -9.099 mm   3.686 cm      10 cm   82.03 MeV     8.431 MeV  1.897 cm  9.028 cm    Shape2   Transportation
G4WT0 > Deposited in scorer: 8.43107 MeV
G4WT0 >     6  -1.117 cm   4.011 cm   11.23 cm   77.68 MeV     3.295 MeV  1.286 cm  10.31 cm   Envelope     hIoni
G4WT0 >
    :----- List of secondaries ----------------
```

To describe a volume's shape, we use the concept of a solid. A solid is a geometrical object that has a shape and specific values for each of that shape's dimensions. A cube with a side of 10 centimeters and a cylinder of radius 30 cm and length 75 cm are examples of solids.

To describe a volume's full properties, we use a logical volume. It includes the geometrical properties of the solid, and adds physical characteristics: the material of the volume; whether it contains any sensitive detector elements; the magnetic field; etc.

We have yet to describe how to position the volume. To do this you create a physical volume, which places a copy of the logical volume inside a larger, containing, volume.

### 2.2.2 Create a Simple Volume

What do you need to do to create a volume?

- Create a solid.
- Create a logical volume, using this solid, and adding other attributes.

Each of the volume types (solid, logical, and physical) has an associated registry (*VolumeStore*) which contains a list of all the objects of that type constructed so far. The registries will automatically delete those objects when requested; users should not deleted geometry objects manually.

### 2.2.3 Choose a Solid

To create a simple box, you only need to define its name and its extent along each of the Cartesian axes.

Listing 2.4: Creating a box.

```
G4double world_hx = 3.0*m;
G4double world_hy = 1.0*m;
G4double world_hz = 1.0*m;

G4Box* worldBox
   = new G4Box("World", world_hx, world_hy, world_hz);
```

This creates a box named “World” with the extent from -3.0 meters to +3.0 meters along the X axis, from -1.0 to 1.0 meters in Y, and from -1.0 to 1.0 meters in Z. Note that the G4Box constructor takes as arguments the halves of the total box size.

It is also very simple to create a cylinder. To do this, you can use the `G4Tubs` class.

Listing 2.5: Creating a cylinder.

```
G4double innerRadius = 0.*cm;
G4double outerRadius = 60.*cm;
G4double hz = 25.*cm;
G4double startAngle = 0.*deg;
G4double spanningAngle = 360.*deg;

G4Tubs* trackerTube
  = new G4Tubs("Tracker",
               innerRadius,
               outerRadius,
               hz,
               startAngle,
               spanningAngle);
```

This creates a full cylinder, named “Tracker”, of radius 60 centimeters and length 50 cm (the hz parameter represents the half length in Z).

### 2.2.4 Create a Logical Volume

To create a logical volume, you must start with a solid and a material. So, using the box created above, you can create a simple logical volume filled with argon gas (see *How to Specify Materials in the Detector*) by entering:

```
G4LogicalVolume* worldLog
   = new G4LogicalVolume(worldBox, Ar, "World");
```

This logical volume is named “World”.

Similarly we create a logical volume with the cylindrical solid filled with aluminium

```
G4LogicalVolume* trackerLog
  = new G4LogicalVolume(trackerTube, Al, "Tracker");
```

and named “Tracker”.

### 2.2.5 Place a Volume

How do you place a volume? You start with a logical volume, and then you decide the already existing volume inside of which to place it. Then you decide where to place its center within that volume, and how to rotate it. Once you have made these decisions, you can create a physical volume, which is the placed instance of the volume, and embodies all of these attributes.

### 2.2.6 Create a Physical Volume

You create a physical volume starting with your logical volume. A physical volume is simply a placed instance of the logical volume. This instance must be placed inside a mother logical volume. For simplicity it is unrotated:

Listing 2.6: A simple physical volume.

```
G4double pos_x = -1.0*meter;
G4double pos_y =  0.0*meter;
G4double pos_z =  0.0*meter;

G4VPhysicalVolume* trackerPhys
  = new G4PVPlacement(0,                         // no rotation
                      G4ThreeVector(pos_x, pos_y, pos_z),
                                                 // translation position
                      trackerLog,                // its logical volume
                      "Tracker",                 // its name
                      worldLog,                  // its mother (logical) volume
                      false,                     // no boolean operations
                      0);                        // its copy number
```

This places the logical volume `trackerLog` at the origin of the mother volume `worldLog`, shifted by one meter along X and unrotated. The resulting physical volume is named “Tracker” and has a copy number of 0.

An exception exists to the rule that a physical volume must be placed inside a mother volume. That exception is for the World volume, which is the largest volume created, and which contains all other volumes. This volume obviously cannot be contained in any other. Instead, it must be created as a `G4PVPlacement` with a null mother pointer. It also must be unrotated, and it must be placed at the origin of the global coordinate system.

Generally, it is best to choose a simple solid as the World volume, the G4Box solid type is used in all basic examples.

### 2.2.7 Coordinate Systems and Rotations

In GEANT4, the rotation matrix associated to a placed physical volume represents the rotation of the reference system of this volume with respect to its mother.

A rotation matrix is normally constructed as in CLHEP, by instantiating the identity matrix and then applying a rotation to it. This is also demonstrated in Example B3.

## 2.3 How to Specify Materials in the Detector

### 2.3.1 General Considerations

In nature, general materials (chemical compounds, mixtures) are made of elements, and elements are made of isotopes. Therefore, these are the three main classes designed in GEANT4. Each of these classes has a table as a static data member, which is for keeping track of the instances created of the respective classes. All three objects automatically register themselves into the corresponding table on construction, and should never be deleted in user code.

The `G4Element` class describes the properties of the atoms:

- atomic number,
- number of nucleons,
- atomic mass,
- shell energy,
- as well as quantities such as cross sections per atom, etc.

The `G4Material` class describes the macroscopic properties of matter:

- density,
- state,
- temperature,
- pressure,
- as well as macroscopic quantities like radiation length, mean free path, dE/dx, etc.

The `G4Material` class is the one which is visible to the rest of the toolkit, and is used by the tracking, the geometry, and the physics. It contains all the information relative to the eventual elements and isotopes of which it is made, at the same time hiding the implementation details.

### 2.3.2 Define a Simple Material

In the example below, liquid argon is created, by specifying its name, density, mass per mole, and atomic number.

Listing 2.7: Creating liquid argon.

```
G4double z, a, density;
density = 1.390*g/cm3;
a = 39.95*g/mole;

G4Material* lAr = new G4Material(name="liquidArgon", z=18., a, density);
```

The pointer to the material, *lAr*, will be used to specify the matter of which a given logical volume is made:

```
G4LogicalVolume* myLbox = new G4LogicalVolume(aBox,lAr,"Lbox",0,0,0);
```

### 2.3.3 Define a Molecule

In the example below, the water, *H2O*, is built from its components, by specifying the number of atoms in the molecule.

Listing 2.8: Creating water by defining its molecular components.

```
G4double z, a, density;
G4String name, symbol;
G4int ncomponents, natoms;

a = 1.01*g/mole;
G4Element* elH  = new G4Element(name="Hydrogen",symbol="H" , z= 1., a);

a = 16.00*g/mole;
G4Element* elO  = new G4Element(name="Oxygen"  ,symbol="O" , z= 8., a);

density = 1.000*g/cm3;
G4Material* H2O = new G4Material(name="Water",density,ncomponents=2);
H2O->AddElement(elH, natoms=2);
H2O->AddElement(elO, natoms=1);
```

### 2.3.4 Define a Mixture by Fractional Mass

In the example below, air is built from nitrogen and oxygen, by giving the fractional mass of each component.

Listing 2.9: Creating air by defining the fractional mass of its components.

```
G4double z, a, fractionmass, density;
G4String name, symbol;
G4int ncomponents;

a = 14.01*g/mole;
G4Element* elN  = new G4Element(name="Nitrogen",symbol="N" , z= 7., a);

a = 16.00*g/mole;
G4Element* elO  = new G4Element(name="Oxygen"  ,symbol="O" , z= 8., a);

density = 1.290*mg/cm3;
G4Material* Air = new G4Material(name="Air  ",density,ncomponents=2);
Air->AddElement(elN, fractionmass=70*perCent);
Air->AddElement(elO, fractionmass=30*perCent);
```

### 2.3.5 Define a Material from the GEANT4 Material Database

In the example below, air and water are accessed via the GEANT4 material database.

Listing 2.10: Defining air and water from the internal GEANT4 database.

```
G4NistManager* man = G4NistManager::Instance();

G4Material* H2O  = man->FindOrBuildMaterial("G4_WATER");
G4Material* Air  = man->FindOrBuildMaterial("G4_AIR");
```

### 2.3.6 Define a Material from the Base Material

It is possible to build new material on base of an existing “base” material. This feature is useful for electromagnetic physics allowing to peak up for the derived material all correction data and precomputed tables of stopping powers and cross sections of the base material. In the example below, two methods how to create water with unusual density are shown.

Listing 2.11: Defining water with user defined density on base of G4_WATER.

```
G4double density;

density = 1.05*mg/cm3;
G4Material* water1 = new G4Material("Water_1.05",density,"G4_WATER");

density = 1.03*mg/cm3;
G4NistManager* man = G4NistManager::Instance();
G4Material* water2 = man->BuildMaterialWithNewDensity("Water_1.03","G4_WATER",density);
```

### 2.3.7 Print Material Information

Listing 2.12: Printing information about materials.

```
G4cout << H2O;                                \\ print a given material
G4cout << *(G4Material::GetMaterialTable());  \\ print the list of materials
```

In GEANT4 examples you all possible ways to build a material.

### 2.3.8 Access to GEANT4 material database

Listing 2.13: GEANT4 material database may be accessed via UI commands.

```
/material/nist/printElement  Fe      \\ print element by name
/material/nist/printElementZ 13      \\ print element by atomic number
/material/nist/listMaterials type    \\ print materials type = [simple | compound | hep | all]
/material/g4/printElement    elmName \\ print instantiated element by name
/material/g4/printMaterial   matName \\ print instantiated material by name
```

In GEANT4 examples you with find all possible ways to build a material.

## 2.4 How to Specify Particles

`G4VUserPhysicsList` is one of the mandatory user base classes described in *How to Define the main() Program*. Within this class all particles and physics processes to be used in your simulation must be defined. The range cut-off parameter should also be defined in this class.

The user must create a class derived from `G4VuserPhysicsList` and implement the following pure virtual methods:

```
ConstructParticle();      // construction of particles
ConstructProcess();       // construct processes and register them to particles
```

The user may also want to override the default implementation of the following virtual method:

```
SetCuts();                // setting a range cut value for all particles
```

This section provides some simple examples of the `ConstructParticle()` and `SetCuts()` methods. For information on `ConstructProcess()` methods, please see *How to Specify Physics Processes*.

## 2.4.1 Particle Definition

GEANT4 provides various types of particles for use in simulations:

- ordinary particles, such as electrons, protons, and gammas
- resonant particles with very short lifetimes, such as vector mesons and delta baryons
- nuclei, such as deuteron, alpha, and heavy ions (including hyper-nuclei)
- quarks, di-quarks, and gluon

Each particle is represented by its own class, which is derived from `G4ParticleDefinition`. (Exception: G4Ions represents all heavy nuclei. Please see *Particles*.) Particles are organized into six major categories:

- lepton,
- meson,
- baryon,
- boson,
- shortlived and
- ion,

each of which is defined in a corresponding sub-directory under `geant4/source/particles`. There is also a corresponding granular library for each particle category.

### The `G4ParticleDefinition` Class

`G4ParticleDefinition` has properties which characterize individual particles, such as, name, mass, charge, spin, and so on. Most of these properties are "read-only" and can not be changed directly. `G4ParticlePropertyTable` is used to retrieve (load) particle property of `G4ParticleDefinition` into (from) `G4ParticlePropertyData`.

### How to Access a Particle

Each particle class type represents an individual particle type, and each class has a single object. This object can be accessed by using the static method of each class. There are some exceptions to this rule; please see *Particles* for details.

For example, the class `G4Electron` represents the electron and the member `G4Electron::theInstance` points its only object. The pointer to this object is available through the static methods `G4Electron::ElectronDefinition()`. `G4Electron::Definition()`.

More than 100 types of particles are provided by default, to be used in various physics processes. In normal applications, users will not need to define their own particles.

The unique object for each particle class is created when its static method to get the pointer is called at the first time. Because particles are dynamic objects and should be instantiated before initialization of physics processes, you must explicitly invoke static methods of all particle classes required by your program at the initialization step. (NOTE: The particle object was static and created automatically before 8.0 release)

### Dictionary of Particles

The G4ParticleTable class is provided as a dictionary of particles. Various utility methods are provided, such as:

```
FindParticle(G4String name);          // find the particle by name
FindParticle(G4int PDGencoding)       // find the particle by PDG encoding .
```

G4ParticleTable is defined as a singleton object, and the static method G4ParticleTable::GetParticleTable() provides its pointer.

As for heavy ions (including hyper-nuclei), objects are created dynamically by requests from users and processes. The G4ParticleTable class provides methods to create ions, such as:

```
G4ParticleDefinition* GetIon(  G4int    atomicNumber,
                               G4int    atomicMass,
                               G4double   excitationEnergy);
```

Particles are registered automatically during construction. The user has no control over particle registration.

### Constructing Particles

ConstructParticle() is a pure virtual method, in which the static member functions for all the particles you require should be called. This ensures that objects of these particles are created.

**Warning:** You must define "ALL PARTICLE TYPES" which are used in your application, except for heavy ions. "ALL PARTICLE TYPES" means not only primary particles, but also all other particles which may appear as secondaries generated by physics processes you use. Beginning with GEANT4 version 8.0, you should keep this rule strictly because all particle definitions are revised to "non-static" objects.

For example, suppose you need a proton and a geantino, which is a virtual particle used for simulation and which does not interact with materials. The ConstructParticle() method is implemented as below:

Listing 2.14: Construct a proton and a geantino.

```
void MyPhysicsList::ConstructParticle()
{
  G4Proton::ProtonDefinition();
  G4Geantino::GeantinoDefinition();
}
```

Due to the large number of pre-defined particles in GEANT4, it is cumbersome to list all the particles by this method. If you want all the particles in a GEANT4 particle category, there are six utility classes, corresponding to each of the particle categories, which perform this function:

- G4BosonConstructor
- G4LeptonConstructor
- G4MesonConstructor
- G4BaryonConstructor
- G4IonConstructor
- G4ShortlivedConstructor.

An example of this is shown in ExN05PhysicsList, listed below.

Listing 2.15: Construct all leptons.

```
void ExN05PhysicsList::ConstructLeptons()
{
  // Construct all leptons
  G4LeptonConstructor pConstructor;
  pConstructor.ConstructParticle();
}
```

## 2.4.2 Range Cuts

To avoid infrared divergence, some electromagnetic processes require a threshold below which no secondary will be generated. Because of this requirement, gammas, electrons and positrons require production threshold. This threshold should be defined as a distance, or range cut-off, which is internally converted to an energy for individual materials. The range threshold should be defined in the initialization phase using the `SetCuts()` method of `G4VUserPhysicsList`. *Cuts per Region* discusses threshold and tracking cuts in detail.

### Setting the cuts

Production threshold values should be defined in `SetCuts()` which is a virtual method of the `G4VUserPhysicsList`. Construction of particles, materials, and processes should precede the invocation of `SetCuts()`. `G4RunManager` takes care of this sequence in usual applications.

This range cut value is converted threshold energies for each material and for each particle type (i.e. electron, positron and gamma) so that the particle with threshold energy stops (or is absorbed) after traveling the range cut distance. In addition, from the 9.3 release ,this range cut value is applied to the proton as production thresholds of nuclei for hadron elastic processes. In this case, the range cut value does not means the distance of traveling. Threshold energies are calculated by a simple formula from the cut in range.

Note that the upper limit of the threshold energy is defined as 10 GeV. If you want to set higher threshold energy, you can change the limit by using "/cuts/setMaxCutEnergy" command before setting the range cut.

The idea of a "unique cut value in range" is one of the important features of GEANT4 and is used to handle cut values in a coherent manner. For most applications, users need to determine only one cut value in range, and apply this value to gammas, electrons and positrons alike. (and proton too)

The default implementation of `SetCuts()` method provides a `defaultCutValue` member as the unique range cut-off value for all particle types. The `defaultCutValue` is set to 1.0 mm by default. User can change this value by `SetDefaultCutValue()` The "/run/setCut" command may be used to change the default cut value interactively.

**Warning:** DO NOT change cut values inside the event loop. Cut values may however be changed between runs.

It is possible to set different range cut values for gammas, electrons and positrons by using `SetCutValue()` methods (or using "/run/setCutForAGivenParticle" command). However, user must be careful with physics outputs because GEANT4 processes (especially energy loss) are designed to conform to the "unique cut value in range" scheme.

Beginning with GEANT4 version 5.1, it is now possible to set production thresholds for each geometrical region. This new functionality is described in *Cuts per Region*.

## 2.5 How to Specify Physics Processes

### 2.5.1 Physics Processes

Physics processes describe how particles interact with materials. GEANT4 provides seven major categories of processes:

- electromagnetic,
- hadronic,
- transportation,
- decay,
- optical,
- photolepton_hadron, and
- parameterisation.

All physics processes are derived from the `G4VProcess` base class. Its virtual methods

- `AtRestDoIt`,
- `AlongStepDoIt`, and
- `PostStepDoIt`

and the corresponding methods

- `AtRestGetPhysicalInteractionLength`,
- `AlongStepGetPhysicalInteractionLength`, and
- `PostStepGetPhysicalInteractionLength`

describe the behavior of a physics process when they are implemented in a derived class. The details of these methods are described in *Physics Processes*.

The following are specialized base classes to be used for simple processes:

**`G4VAtRestProcess`** Processes with only `AtRestDoIt`
**`G4VContinuousProcess`** Processes with only `AlongStepDoIt`
**`G4VDiscreteProcess`** processes with only `PostStepDoIt`

Another 4 virtual classes, such as `G4VContinuousDiscreteProcess`, are provided for complex processes.

### 2.5.2 Managing Processes

The `G4ProcessManager` class contains a list of processes that a particle can undertake. It has information on the order of invocation of the processes, as well as which kind of `DoIt` method is valid for each process in the list. A `G4ProcessManager` object corresponds to each particle and is attached to the `G4ParticleDefiniton` class.

In order to validate processes, they should be registered with the particle's `G4ProcessManager`. Process ordering information is included by using the `AddProcess()` and `SetProcessOrdering()` methods. For registration of simple processes, the `AddAtRestProcess()`, `AddContinuousProcess()` and `AddDiscreteProcess()` methods may be used.

`G4ProcessManager` is able to turn some processes on or off during a run by using the `ActivateProcess()` and `InActivateProcess()` methods. These methods are valid only after process registration is complete, so they must not be used in the *PreInit* phase.

The `G4VUserPhysicsList` class creates and attaches `G4ProcessManager` objects to all particle classes defined in the `ConstructParticle()` method.

## 2.5.3 Specifying Physics Processes

`G4VUserPhysicsList` is the base class for a "mandatory user class" (see *How to Define the main() Program*), in which all physics processes and all particles required in a simulation must be registered. The user must create a class derived from `G4VUserPhysicsList` and implement the pure virtual method `ConstructProcess()`.

For example, if just the `G4Geantino` particle class is required, only the transportation process need be registered. The `ConstructProcess()` method would then be implemented as follows:

Listing 2.16: Register processes for a geantino.

```
void MyPhysicsList::ConstructProcess()
{
  // Define transportation process
  AddTransportation();
}
```

Here, the `AddTransportation()` method is provided in the `G4VUserPhysicsList` class to register the `G4Transportation` class with all particle classes. The `G4Transportation` class (and/or related classes) describes the particle motion in space and time. It is the mandatory process for tracking particles.

In the `ConstructProcess()` method, physics processes should be created and registered with each particle's instance of `G4ProcessManager`.

An example of process registration is given in the `G4VUserPhysicsList::AddTransportation()` method.

Registration in `G4ProcessManager` is a complex procedure for other processes and particles because the relations between processes are crucial for some processes. In order to ease registration procedures, G4PhysicsListHelper is provided. Users do not care about type of processes (i.e. AtRest and/or Discrete and/or Continuous ) or ordering parameters.

An example of electromagnetic process registration for the gamma is shown below

Listing 2.17: Register processes for a gamma.

```
void MyPhysicsList::ConstructProcess()
{
  // Define transportation process
  AddTransportation();
  // electromagnetic processes
  ConstructEM();
}

void MyPhysicsList::ConstructEM()
{
  // Get pointer to G4PhysicsListHelper
  G4PhysicsListHelper* ph = G4PhysicsListHelper::GetPhysicsListHelper();

  //  Get pointer to gamma
  G4ParticleDefinition* particle = G4Gamma::GammaDefinition();

  // Construct and register processes for gamma
  ph->RegisterProcess(new G4PhotoElectricEffect(), particle);
  ph->RegisterProcess(new G4ComptonScattering(), particle);
  ph->RegisterProcess(new G4GammaConversion(), particle);
  ph->RegisterProcess(new G4RayleighScattering(), particle);
}
```

## 2.6 How to Generate a Primary Event

### 2.6.1 Generating Primary Events

`G4VuserPrimaryGeneratorAction` is one of the mandatory classes available for deriving your own concrete class. In your concrete class, you have to specify how a primary event should be generated. Actual generation of primary particles will be done by concrete classes of `G4VPrimaryGenerator`, explained in the following subsection. Your `G4VUserPrimaryGeneratorAction` concrete class just arranges the way primary particles are generated.

Listing 2.18: PrimaryGeneratorAction: An example of a G4VUserPrimaryGeneratorAction concrete class using G4ParticleGun. For the usage of G4Particle Gun refer to the next subsection.

```
//////////////////////////////////
// PrimaryGeneratorAction.hh
//////////////////////////////////

#ifndef PrimaryGeneratorAction_h
#define PrimaryGeneratorAction_h 1

#include "G4VUserPrimaryGeneratorAction.hh"
#include "G4ThreeVector.hh"
#include "globals.hh"

class G4ParticleGun;
class G4Event;

class PrimaryGeneratorAction : public G4VUserPrimaryGeneratorAction
{
  public:
    PrimaryGeneratorAction(
      const G4String& particleName = "geantino",
      G4double energy = 1.*MeV,
```

(continues on next page)

(continued from previous page)

```
      G4ThreeVector position= G4ThreeVector(0,0,0),
      G4ThreeVector momentumDirection = G4ThreeVector(0,0,1));
    ~PrimaryGeneratorAction();

    // methods
    virtual void GeneratePrimaries(G4Event*);

  private:
    // data members
    G4ParticleGun*  fParticleGun; //pointer a to G4 service class
};

#endif

//////////////////////////////////
// PrimaryGeneratorAction.cc
//////////////////////////////////

#include "PrimaryGeneratorAction.hh"

#include "G4Event.hh"
#include "G4ParticleGun.hh"
#include "G4ParticleTable.hh"
#include "G4ParticleDefinition.hh"

//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo........oooOO0OOooo......

PrimaryGeneratorAction::PrimaryGeneratorAction(
                                const G4String& particleName,
                                G4double energy,
                                G4ThreeVector position,
                                G4ThreeVector momentumDirection)
  : G4VUserPrimaryGeneratorAction(),
    fParticleGun(0)
{
  G4int nofParticles = 1;
  fParticleGun  = new G4ParticleGun(nofParticles);

  // default particle kinematic
  G4ParticleTable* particleTable = G4ParticleTable::GetParticleTable();
  G4ParticleDefinition* particle
    = particleTable->FindParticle(particleName);
  fParticleGun->SetParticleDefinition(particle);
  fParticleGun->SetParticleEnergy(energy);
  fParticleGun->SetParticlePosition(position);
  fParticleGun->SetParticleMomentumDirection(momentumDirection);
}

//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo........oooOO0OOooo......

PrimaryGeneratorAction::~PrimaryGeneratorAction()
{
  delete fParticleGun;
}

//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo........oooOO0OOooo......

void PrimaryGeneratorAction::GeneratePrimaries(G4Event* anEvent)
{
  // this function is called at the beginning of event

  fParticleGun->GeneratePrimaryVertex(anEvent);
}

//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo........oooOO0OOooo......
```

#### Selection of the generator

In the constructor of your `G4VUserPrimaryGeneratorAction`, you should instantiate the primary generator(s). If necessary, you need to set some initial conditions for the generator(s).

In *PrimaryGeneratorAction*, `G4ParticleGun` is constructed to use as the actual primary particle generator. Methods of `G4ParticleGun` are described in the following section. Please note that the primary generator object(s) you construct in your `G4VUserPrimaryGeneratorAction` concrete class must be deleted in your destructor.

#### Generation of an event

`G4VUserPrimaryGeneratorAction` has a pure virtual method named `generatePrimaries()`. This method is invoked at the beginning of each event. In this method, you have to invoke the `G4VPrimaryGenerator` concrete class you instantiated via the `generatePrimaryVertex()` method.

You can invoke more than one generator and/or invoke one generator more than once. Mixing up several generators can produce a more complicated primary event.

### 2.6.2 G4VPrimaryGenerator

GEANT4 provides three `G4VPrimaryGenerator` concrete classes. Among these `G4ParticleGun` and `G4GeneralParticleSource` will be discussed here. The third one is `G4HEPEvtInterface`, which will be discussed in *Event Generator Interface*.

#### G4ParticleGun

`G4ParticleGun` is a generator provided by GEANT4. This class generates primary particle(s) with a given momentum and position. It does not provide any sort of randomizing. The constructor of `G4ParticleGun` takes an integer which causes the generation of one or more primaries of exactly same kinematics. It is a rather frequent user requirement to generate a primary with randomized energy, momentum, and/or position. Such randomization can be achieved by invoking various set methods provided by `G4ParticleGun`. The invocation of these methods should be implemented in the `generatePrimaries()` method of your concrete `G4VUserPrimaryGeneratorAction` class before invoking `generatePrimaryVertex()` of `G4ParticleGun`. GEANT4 provides various random number generation methods with various distributions (see *Global Usage Classes*).

#### Public methods of `G4ParticleGun`

The following methods are provided by `G4ParticleGun`, and all of them can be invoked from the `generatePrimaries()` method in your concrete `G4VUserPrimaryGeneratorAction` class.

- `void SetParticleDefinition(G4ParticleDefinition*)`
- `void SetParticleMomentum(G4ParticleMomentum)`
- `void SetParticleMomentumDirection(G4ThreeVector)`
- `void SetParticleEnergy(G4double)`
- `void SetParticleTime(G4double)`
- `void SetParticlePosition(G4ThreeVector)`
- `void SetParticlePolarization(G4ThreeVector)`
- `void SetNumberOfParticles(G4int)`

### G4GeneralParticleSource

For many applications G4ParticleGun is a suitable particle generator. However if you want to generate primary particles in more sophisticated manner, you can utilize G4GeneralParticleSource, the GEANT4 General Particle Source module (GPS), discussed in the next section ( *General Particle Source*).

## 2.7 GEANT4 General Particle Source

### 2.7.1 Introduction

The G4GeneralParticleSource (GPS) is part of the GEANT4 toolkit for Monte-Carlo, high-energy particle transport. Specifically, it allows the specifications of the spectral, spatial and angular distribution of the primary source particles. An overview of the GPS class structure is presented here. *Configuration* covers the configuration of GPS for a user application, and *Macro Commands* describes the macro command interface. *Example Macro File* gives an example input file to guide the first time user.

G4GeneralParticleSource is used exactly the same way as G4ParticleGun in a GEANT4 application. In existing applications one can simply change your PrimaryGeneratorAction by globally replacing G4ParticleGun with G4GeneralParticleSource. GPS may be configured via command line, or macro based, input. The experienced user may also hard-code distributions using the methods and classes of the GPS that are described in more detail in a technical note[1].

The class diagram of GPS is shown in Fig. 2.1. As of version 10.01, a split-class mechanism was introduced to reduce memory usage in multithreaded mode. The G4GeneralParticleSourceData class is a thread-safe singleton which provides access to the source information for the G4GeneralParticleSource class. The G4GeneralParticleSourceData class can have multiple instantiations of the G4SingleParticleSource class, each with independent positional, angular and energy distributions as well as incident particle types. To the user, this change should be transparent.

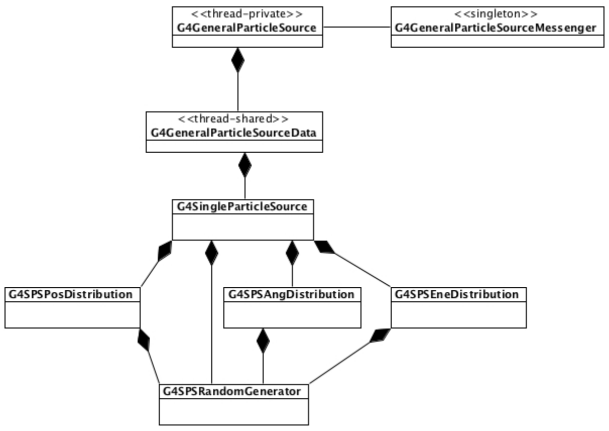


Fig. 2.1: The class diagram of G4GeneralParticleSource.

[1] General purpose Source Particle Module for GEANT4/SPARSET: Technical Note, UoS-GSPM-Tech, Issue 1.1, C Ferguson, February 2000.

### 2.7.2 Configuration

GPS allows the user to control the following characteristics of primary particles:

- Spatial sampling: on simple 2D or 3D surfaces such as discs, spheres, and boxes.
- Angular distribution: unidirectional, isotropic, cosine-law, beam or arbitrary (user defined).
- Spectrum: linear, exponential, power-law, Gaussian, blackbody, or piece-wise fits to data.
- Multiple sources: multiple independent sources can be used in the same run.

As noted above, `G4GeneralParticleSource` is used exactly the same way as `G4ParticleGun` in a GEANT4 application, and may be substituted for the latter by "global search and replace" in existing application source code.

#### Position Distribution

The position distribution can be defined by using several basic shapes to contain the starting positions of the particles. The easiest source distribution to define is a point source. One could also define planar sources, where the particles emanate from circles, annuli, ellipses, squares or rectangles. There are also methods for defining 1D or 2D accelerator beam spots. The five planes are oriented in the x-y plane. To define a circle one gives the radius, for an annulus one gives the inner and outer radii, and for an ellipse, a square or a rectangle one gives the half-lengths in x and y.

More complicated still, one can define surface or volume sources where the input particles can be confined to either the surface of a three dimensional shape or to within its entire volume. The four 3D shapes used within G4GeneralParticleSource are sphere, ellipsoid, cylinder and parallelepiped. A sphere can be defined simply by specifying the radius. Ellipsoids are defined by giving their half-lengths in x, y and z. Cylinders are defined such that the axis is parallel to the z-axis, the user is therefore required to give the radius and the z half-length. Parallelepipeds are defined by giving x, y and z half-lengths, plus the angles $\alpha$, $\theta$, and $\phi$ (Fig. 2.2).

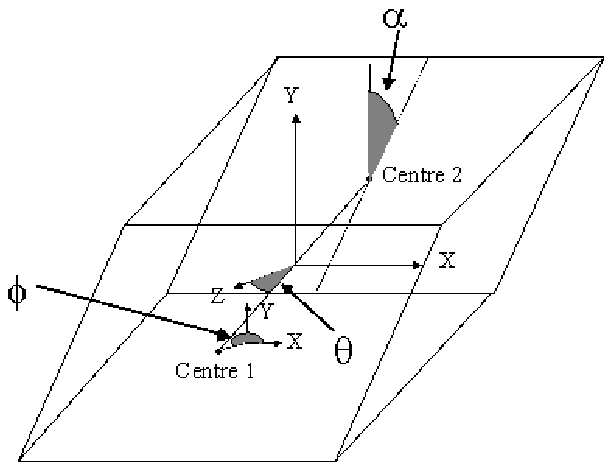


Fig. 2.2: The angles used in the definition of a Parallelepiped.

To allow easy definition of the sources, the planes and shapes are assumed to be orientated in a particular direction to the coordinate axes, as described above. For more general applications, the user may supply two vectors (x' and a vector in the plane x'-y') to rotate the co-ordinate axes of the shape with respect to the overall co-ordinate system (Fig. 2.3). The rotation matrix is automatically calculated within G4GeneralParticleSource. The starting points of particles are always distributed homogeneously over the 2D or 3D surfaces, although biasing can change this.

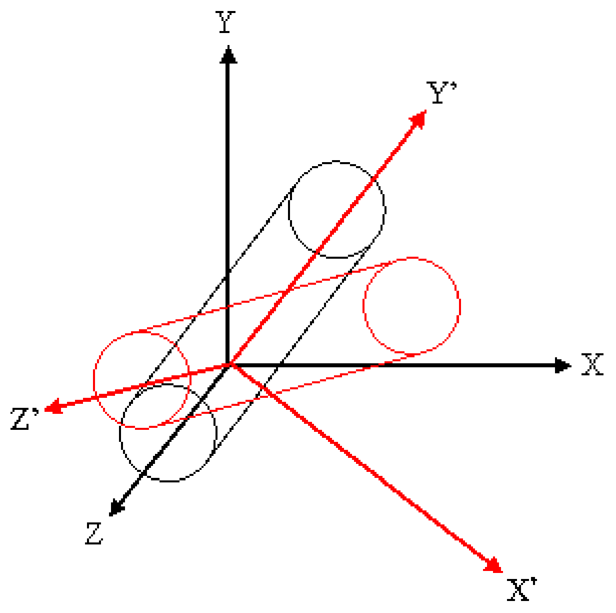


Fig. 2.3: An illustration of the use of rotation matrices. A cylinder is defined with its axis parallel to the z-axis (black lines), but the definition of 2 vectors can rotate it into the frame given by x', y', z' (red lines).

### Angular Distribution

The angular distribution is used to control the directions in which the particles emanate from/incident upon the source point. In general there are three main choices, isotropic, cosine-law or user-defined. In addition there are options for specifying parallel beam as well as diverse accelerator beams. The isotropic distribution represents what would be seen from a uniform $4\pi$ flux. The cosine-law represents the distribution seen at a plane from a uniform $2\pi$ flux.

It is possible to bias (*Biasing*) both $\theta$ and $\phi$ for any of the predefined distributions, including setting lower and upper limits to $\theta$ and $\phi$. User-defined distributions cannot be additionally biased (any bias should obviously be incorporated into the user definition).

Incident with zenith angle $\theta = 0$ means the particle is travelling along the -z axis. It is important to bear this in mind when specifying user-defined co-ordinates for angular distributions. The user must be careful to rotate the co-ordinate axes of the angular distribution if they have rotated the position distribution (Fig. 2.3).

The user defined distribution requires the user to enter a histogram in either $\theta$ or $\phi$ or both. The user-defined distribution may be specified either with respect to the coordinate axes or with respect to the surface-normal of a shape or volume. For the surface-normal distribution, $\theta$ should only be defined between 0 and $\pi/2$, not the usual 0 to $\pi$ range.

The top-level `/gps/direction` command uses direction cosines to specify the primary particle direction, as follows:

$$\begin{aligned} P_x &= -\sin\theta\cos\phi \\ P_y &= -\sin\theta\sin\phi \\ P_z &= -\cos\theta \end{aligned} \tag{2.1}$$

### Energy Distribution

The energy of the input particles can be set to follow several built-in functions or a user-defined one, as shown in Table 2.1. The user can bias any of the pre-defined energy distributions in order to speed up the simulation (user-defined distributions are already biased, by construction).

Table 2.1: Energy distribution commands.

| Spectrum | Abbreviation | Functional Form | User Parameters |
|---|---|---|---|
| mono-energetic | Mono | $I \propto \delta(E - E_0)$ | Energy $E_0$ |
| linear | Lin | $I \propto I_0 + m \times E$ | Intercept $I_0$, slope $m$ |
| exponential | Exp | $I \propto \exp(-E/E_0)$ | Energy scale-height $E_0$ |
| power-law | Pow | $I \propto E^{\alpha}$ | Spectral index $\alpha$ |
| Gaussian | Gauss | $I = (2\pi\sigma)^{-\frac{1}{2}} \exp[-(E/E_0)^2/\sigma^2]$ | Mean energy $E_0$, standard deviation $\sigma$ |
| bremsstrahlung | Brem | $I = \int 2E^2[h^2c^2(\exp(-E/kT) - 1)]^{-1}$ | Temperature $T$ |
| black body | Bbody | $I \propto (kT)^{\frac{1}{2}} E \exp(-E/kT)$ | Temperature $T$ (see text) |
| cosmic diffuse gamma ray | Cdg | $I \propto [(E/E_b)^{\alpha_1} + (E/E_b)^{\alpha_2}]^{-1}$ | Energy range $E_{\min}$ to $E_{\max}$; energy $E_b$ and indices $\alpha_1$ and $\alpha_2$ are fixed (see text) |

There is also the option for the user to define a histogram in energy (“User”) or energy per nucleon (“Epn”) or to give an arbitrary point-wise spectrum (“Arb”) that can be fit with various simple functions. The data for histograms or point spectra must be provided in ascending bin (abscissa) order. The point-wise spectrum may be differential (as with a binned histogram) or integral (a cumulative distribution function). If integral, the data must satisfy $s(e1) \geq s(e2)$ for $e1 < e2$ when entered; this is not validated by the GPS code. The maximum energy of an integral spectrum is defined by the last-but-one data point, because GPS converts to a differential spectrum internally.

Unlike the other spectral distributions it has proved difficult to integrate indefinitely the black-body spectrum and this has lead to an alternative approach. Instead it has been decided to use the black-body formula to create a 10,000 bin histogram and then to produce random energies from this.

Similarly, the broken power-law for cosmic diffuse gamma rays makes generating an indefinite integral CDF problematic. Instead, the minimum and maximum energies specified by the user are used to construct a definite-integral CDF from which random energies are selected.

### Biasing

The user can bias distributions by entering a histogram. It is the random numbers from which the quantities are picked that are biased and so one only needs a histogram from 0 to 1. Great care must be taken when using this option, as the way a quantity is calculated will affect how the biasing works, as discussed below. Bias histograms are entered in the same way as other user-defined histograms.

When creating biasing histograms it is important to bear in mind the way quantities are generated from those numbers. For example let us compare the biasing of a $\theta$ distribution with that of a $\phi$ distribution. Let us divide the $\theta$ and $\phi$ ranges up into 10 bins, and then decide we want to restrict the generated values to the first and last bins. This gives a new $\phi$ range of 0 to 0.628 and 5.655 to 6.283. Since $\phi$ is calculated using $\phi = 2\pi \times \mathrm{RNDM}$, this simple biasing will work correctly.

If we now look at $\theta$, we expect to select values in the two ranges 0 to 0.314 (for $0 \leq \mathrm{RNDM} \leq 0.1$) and 2.827 to 3.142 (for $0 \leq \mathrm{RNDM} \leq 0.9$). However, the polar angle $\theta$ is calculated from the formula $\theta = \arccos(1 - 2 \times \mathrm{RNDM})$. From this, we see that 0.1 gives a $\theta$ of 0.644 and a $\mathrm{RNDM}$ of 0.9 gives a $\theta$ of 2.498. This means that the above will not bias the distribution as the user had wished. The user must therefore take into account the method used to generate random quantities when trying to apply a biasing scheme to them. Some quantities such as x, y, z and $\phi$ will be relatively easy to bias, but others may require more thought.

### User-Defined Histograms

The user can define histograms for several reasons: angular distributions in either $\theta$ or $\phi$; energy distributions; energy per nucleon distributions; or biasing of x, y, z, $\theta$, $\phi$, or energy. Even though the reasons may be different the approach is the same.

To choose a histogram the command `/gps/hist/type` is used (*Macro Commands*). If one wanted to enter an angular distribution one would type "theta" or "phi" as the argument. The histogram is loaded, one bin at a time, by using the `/gps/hist/point` command, followed by its two arguments the upper boundary of the bin and the weight (or area) of the bin. Histograms are therefore differential functions.

Currently histograms are limited to 1024 bins. The first value of each user input data pair is treated as the upper edge of the histogram bin and the second value is the bin content. The exception is the very first data pair the user input whose first value is the treated as the lower edge of the first bin of the histogram, and the second value is not used. This rule applies to all distribution histograms, as well as histograms for biasing.

The user has to be aware of the limitations of histograms. For example, in general $\theta$ is defined between 0 and $\pi$ and $\phi$ is defined between 0 and $2\pi$, so histograms defined outside of these limits may not give the user what they want (see also *Biasing*).

## 2.7.3 Macro Commands

`G4GeneralParticleSource` can be configured by typing commands from the `/gps` command directory tree, or including the `/gps` commands in a g4macro file.

### `G4ParticleGun` equivalent commands

Table 2.2: `G4ParticleGun` equivalent commands.

| Command | Arguments | Description and restrictions |
|---|---|---|
| /gps/List | | List available incident particles |
| /gps/particle | name | Defines the particle type [default *geantino*], using GEANT4 naming convention. |
| /gps/direction | Px Py Pz | Set the momentum direction [default (1,0,0)] of generated particles using (2.1) |
| /gps/energy | E unit | Sets the energy [default 1 MeV] for mono-energetic sources. The units can be eV, keV, MeV, GeV, TeV or PeV. (NB: it is recommended to use /gps/ene/mono instead.) |
| /gps/position | X Y Z unit | Sets the centre co-ordinates (X,Y,Z) of the source [default (0,0,0) cm]. The units can be micron, mm, cm, m or km. (NB: it is recommended to use /gps/pos/centre instead.) |
| /gps/ion | Z A Q E | After `/gps/particle ion`, sets the properties (atomic number Z, atomic mass A, ionic charge Q, excitation energy E in keV) of the ion. |
| /gps/ionLvl | Z A Q lvl | After `/gps/particle ion`, sets the properties (atomic number Z, atomic mass A, ionic charge Q, Number of metastable state excitation level (0-9) of the ion. |
| /gps/time | t0 unit | Sets the primary particle (event) time [default 0 ns]. The units can be ps, ns, us, ms, or s. |
| /gps/polarization | Px Py Pz | Sets the polarization vector of the source, which does not need to be a unit vector. |
| /gps/number | N | Sets the number of particles [default 1] to simulate on each event. |
| /gps/verbose | level | Control the amount of information printed out by the GPS code. Larger values produce more detailed output. |

### Multiple source specification

Table 2.3: Multiple source specification.

| Command | Arguments | Description and restrictions |
|---|---|---|
| /gps/source/add | intensity | Add a new particle source with the specified intensity |
| /gps/source/list | | List the particle sources defined. |
| /gps/source/clear | | Remove all defined particle sources. |
| /gps/source/show | | Display the current particle source |
| /gps/source/set | index | Select the specified particle source as the current one. |
| /gps/source/delete | index | Remove the specified particle source. |
| /gps/source/ multiplevertex | flag | Specify *true* for simultaneous generation of multiple vertices, one from each specified source. False [default] generates a single vertex, choosing one source randomly. |
| /gps/source/ intensity | intensity | Reset the current source to the specified intensity |
| /gps/source/ flatsampling | flag | Set to True to allow biased sampling among the sources. Setting to True will ignore source intensities. The default is False. |

### Source position and structure

Table 2.4: Source position and structure.

| Command | Arguments | Description and restrictions |
|---|---|---|
| /gps/pos/type | dist | Sets the source positional distribution type: *Point* [default], *Plane, Beam*, *Surface, Volume*. |
| /gps/pos/shape | shape | Sets the source shape type, after `/gps/pos/type` has been used. For a Plane this can be *Circle, Annulus*, *Ellipse, Square, Rectangle*. For both Surface or Volume sources this can be *Sphere, Ellipsoid, Cylinder, Para* (parallelepiped). |
| /gps/pos/centre | X Y Z unit | Sets the centre co-ordinates (X,Y,Z) of the source [default (0,0,0) cm]. The units can only be micron, mm, cm, m or km. |
| /gps/pos/rot1 | $R1_x$ $R1_y$ $R1_z$ | Defines the first (x' direction) vector R1 [default (1,0,0)], which does not need to be a unit vector, and is used together with `/gps/pos/rot2` to create the rotation matrix of the shape defined with `/gps/shape`. |
| /gps/pos/rot2 | $R2_x$ $R2_y$ $R2_z$ | Defines the second vector R2 in the xy plane [default (0,1,0)], which does not need to be a unit vector, and is used together with `/gps/pos/rot1` to create the rotation matrix of the shape defined with `/gps/shape`. |
| /gps/pos/halfx | len unit | Sets the half-length in x [default 0 cm] of the source. The units can only be micron, mm, cm, m or km. |
| /gps/pos/halfy | len unit | Sets the half-length in y [default 0 cm] of the source. The units can only be micron, mm, cm, m or km. |
| /gps/pos/halfz | len unit | Sets the half-length in z [default 0 cm] of the source. The units can only be micron, mm, cm, m or km. |
| /gps/pos/radius | len unit | Sets the radius [default 0 cm] of the source or the outer radius for annuli. The units can only be micron, mm, cm, m or km. |
| /gps/pos/inner_radius | len unit | Sets the inner radius [default 0 cm] for annuli. The units can only be micron, mm, cm, m or km. |
| /gps/pos/sigma_r | sigma unit | Sets the transverse (radial) standard deviation [default 0 cm] of beam position profile. The units can only be micron, mm, cm, m or km. |
| /gps/pos/sigma_x | sigma unit | Sets the standard deviation [default 0 cm] of beam position profile in x-direction. The units can only be micron, mm, cm, m or km. |
| /gps/pos/sigma_y | sigma unit | Sets the standard deviation [default 0 cm] of beam position profile in y-direction. The units can only be micron, mm, cm, m or km. |
| /gps/pos/paralp | alpha unit | Used with a Parallelepiped. The angle [default 0 rad] $\alpha$ formed by the y-axis and the plane joining the centre of the faces parallel to the zx plane at y and +y. The units can only be deg or rad. |
| /gps/pos/parthe | theta unit | Used with a Parallelepiped. Polar angle [default 0 rad] $\theta$ of the line connecting the centre of the face at z to the centre of the face at +z. The units can only be deg or rad. |
| /gps/pos/parphi | phi unit | Used with a Parallelepiped. The azimuth angle [default 0 rad] $\phi$ of the line connecting the centre of the face at z with the centre of the face at +z. The units can only be deg or rad. |
| /gps/pos/confine | name | Allows the user to confine the source to the physical volume *name* [default NULL]. |

### Source direction and angular distribution

Table 2.5: Source direction and angular distribution.

| Command | Arguments | Description and restrictions |
|---|---|---|
| /gps/ang/type | AngDis | Sets the angular distribution type (*iso* [default], *cos, planar, beam1d, beam2d, focused, user*) to either isotropic, cosine-law or user-defined. |
| /gps/ang/rot1 | $AR1_x$ $AR1_y$ $AR1_z$ | Defines the first (x' direction) rotation vector AR1 [default (1,0,0)] for the angular distribution and is not necessarily a unit vector. Used with `/gps/ang/rot2` to compute the angular distribution rotation matrix. |
| /gps/ang/rot2 | $AR2_x$ $AR2_y$ $AR2_z$ | Defines the second rotation vector AR2 in the xy plane [default (0,1,0)] for the angular distribution, which does not necessarily have to be a unit vector. Used with `/gps/ang/rot2` to compute the angular distribution rotation matrix. |
| /gps/ang/mintheta | MinTheta unit | Sets a minimum value [default 0 rad] for the $\theta$ distribution. Units can be deg or rad. |
| /gps/ang/maxtheta | MaxTheta unit | Sets a maximum value [default $\pi$ rad] for the $\theta$ distribution. Units can be deg or rad. |
| /gps/ang/minphi | MinPhi unit | Sets a minimum value [default 0 rad] for the $\phi$ distribution. Units can be deg or rad. |
| /gps/ang/maxphi | MaxPhi unit | Sets a maximum value [default $2\pi$ rad] for the $\phi$ distribution. Units can be deg or rad. |
| /gps/ang/sigma_r | sigma unit | Sets the standard deviation [default 0 rad] of beam directional profile in radial. The units can only be deg or rad. |
| /gps/ang/sigma_x | sigma unit | Sets the standard deviation [default 0 rad] of beam directional profile in x-direction. The units can only be deg or rad. |
| /gps/ang/sigma_y | sigma unit | Sets the standard deviation [default 0 rad] of beam directional profile in y-direction. The units can only be deg or rad. |
| /gps/ang/focuspoint | X Y Z unit | Set the focusing point (X,Y,Z) for the beam [default (0,0,0) cm]. The units can only be micron, mm, cm, m or km. |
| /gps/ang/user_coor | bool | Calculate the angular distribution with respect to the user defined co-ordinate system (*true*), or with respect to the global co-ordinate system (*false*, default). |
| /gps/ang/surfnorm | bool | Allows user to choose whether angular distributions are with respect to the co-ordinate system (*false*, default) or surface normals (*true*) for user-defined distributions. |

### Energy spectra

Table 2.6: Source energy spectra.

| Command | Arguments | Description and restrictions |
|---|---|---|
| /gps/ene/type | EnergyDis | Sets the energy distribution type to one of (see Table Table 2.1): *Mono* (mono-energetic, default), *Lin* (linear), *Pow* (power-law), `Exp` (exponential), *Gauss* (Gaussian), *Brem* (bremsstrahlung), *Bbody* (black-body), *Cdg* (cosmic diffuse gamma-ray), *User* (user-defined histogram), *Arb* (point-wise spectrum), *Epn* (energy-per-nucleon histogram) |
| /gps/ene/min | Emin unit | Sets the minimum [default 0 keV] for the energy distribution. The units can be eV, keV, MeV, GeV, TeV or PeV. |
| /gps/ene/max | Emax unit | Sets the maximum [default 0 keV] for the energy distribution. The units can be eV, keV, MeV, GeV, TeV or PeV. |
| /gps/ene/mono | E unit | Sets the energy [default 1 MeV] for mono-energetic sources. The units can be eV, keV, MeV, GeV, TeV or PeV. |
| /gps/ene/sigma | sigma unit | Sets the standard deviation [default 0 keV] in energy for Gaussian or Mono energy distributions. The units can be eV, keV, MeV, GeV, TeV or PeV. |
| /gps/ene/alpha | alpha | Sets the exponent $\alpha$ [default 0] for a power-law distribution. |
| /gps/ene/temp | T | Sets the temperature in kelvins [default 0] for black body and bremsstrahlung spectra. |
| /gps/ene/ezero | E0 | Sets scale $E_0$ [default 0] for exponential distributions. |
| /gps/ene/gradient | gradient | Sets the gradient (slope) [default 0] for linear distributions. |
| /gps/ene/intercept | intercept | Sets the Y-intercept [default 0] for the linear distributions. |
| /gps/ene/biasAlpha | alpha | Sets the exponent $\alpha$ [default 0] for a biased power-law distribution. Bias weight is determined from the power-law probability distribution. |
| /gps/ene/calculate | | Prepares integral PDFs for the internally-binned cosmic diffuse gamma ray (*Cdg*) and black body (*Bbody*) distributions. |
| /gps/ene/emspec | bool | Allows user to specify distributions are in momentum (*false*) or energy (*true*, default). Only valid for *User* and *Arb* distributions. |
| /gps/ene/diffspec | bool | Allows user to specify whether a point-wise spectrum is integral (*false*) or differential (*true*, default). The integral spectrum is only usable for *Arb* distributions. |

### User-defined histograms and interpolated functions

Table 2.7: User defined histograms and interpolated functions.

| Command | Arguments | Description and restrictions |
|---|---|---|
| /gps/hist/type | type | Set the histogram type: predefined *biasx* [default], *biasy, biasz, biast* (angle $\theta$, *biasp* (angle $\phi$), *biaspt* (position $\theta$, *biaspp* (position $\phi$), *biase*; user-defined histograms *theta, phi, energy, arb* (point-wise), *epn* (energy per nucleon). |
| /gps/hist/reset | type | Re-set the specified histogram: *biasx* [default], , *biasy, biasz, biast*, *biasp, biaspt, biaspp, biase, theta*, *phi, energy, arb, epn*. |
| /gps/hist/point | $E_{hi}$ Weight | Specify one entry (with contents *Weight*) in a histogram (where $E_{hi}$ is the bin upper edge) or point-wise distribution (where $E_{hi}$ is the abscissa). The abscissa $E_{hi}$ must be in GEANT4 default units (MeV for energy, rad for angle). |
| /gps/hist/file | HistFile | Import an arbitrary energy histogram in an ASCII file. The format should be one $E_{hi}$ *Weight* pair per line of the file, following the detailed instructions in *User-defined histograms and interpolated functions* For histograms, $E_{hi}$ is the bin upper edge, for point-wise distributions $E_{hi}$ is the abscissa. The abscissa $E_{hi}$ must be in GEANT4 default units (MeV for energy, rad for angle). |
| /gps/hist/inter | type | Sets the interpolation type (*Lin* linear, *Log* logarithmic, *Exp* exponential, *Spline* cubic spline) for point-wise spectra. This command **must** be issued immediately after the last data point. |

## 2.7.4 Example Macro File

```
# Macro test2.g4mac
/control/verbose 0
/tracking/verbose 0
/event/verbose 0
/gps/verbose 2
/gps/particle gamma
/gps/pos/type Plane
/gps/pos/shape Square
/gps/pos/centre 1 2 1 cm
/gps/pos/halfx 2 cm
/gps/pos/halfy 2 cm
/gps/ang/type cos
/gps/ene/type Lin
/gps/ene/min 2 MeV
/gps/ene/max 10 MeV
/gps/ene/gradient 1
/gps/ene/intercept 1
/run/beamOn 10000
```

The above macro defines a planar source, square in shape, 4 cm by 4 cm and centred at (1,2,1) cm. By default the normal of this plane is the z-axis. The angular distribution is to follow the cosine-law. The energy spectrum is linear, with gradient and intercept equal to 1, and extends from 2 to 10 MeV. 10,000 primaries are to be generated.

The standard GEANT4 output should show that the primary particles start from between 1, 0, 1 and 3, 4, 1 (in cm) and have energies between 2 and 10 MeV, as shown in Fig. 2.4, in which we plotted the actual energy, position and angular distributions of the primary particles generated by the above macro file.

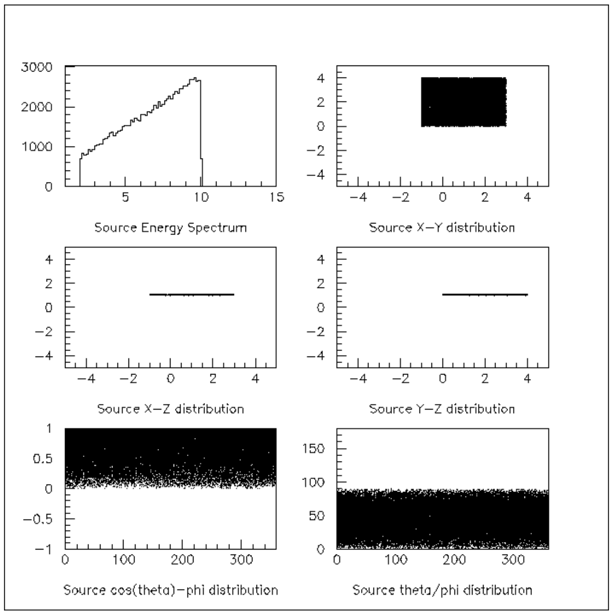


Fig. 2.4: Energy, position and angular distributions of the primary particles as generated by the macro file shown above.

## 2.8 How to Make an Executable Program

The code for the user examples in Geant4 is placed in the subdirectory `examples` of the main Geant4 source package. This directory is installed to the `share/Geant4-G4VERSION/examples` (where `G4VERSION` is the Geant4 version number) subdirectory under the installation prefix. In the following section, a quick overview will be given on how to build a concrete example, "ExampleB1", which is part of the Geant4 distribution, using CMake.

### 2.8.1 Using CMake to Build Applications

Geant4 installs a file named `Geant4Config.cmake` located in

```
+- CMAKE_INSTALL_PREFIX
   +- lib/
      +- cmake/
         +- Geant4/
            +- Geant4Config.cmake
```

which is designed for use with the CMake `find_package` command. Building a Geant4 application using CMake therefore involves writing a `CMakeLists.txt` script using this and other CMake commands to locate Geant4 and describe the build of your client application. Whilst it requires a bit of effort to write the script, CMake provides a very friendly yet powerful tool, especially if you are working on multiple platforms. It is therefore the method we recommend for building Geant4 applications.

We'll use Basic Example B1, which you may find in the Geant4 source directory under `examples/basic/B1`, to demonstrate the use of CMake to build a Geant4 application. You'll find links to the latest CMake documentation for the commands used throughout, so please follow these for further information. The application sources and scripts are arranged in the following directory structure:

```
+- B1/
   +- CMakeLists.txt
   +- exampleB1.cc
```

(continues on next page)

(continued from previous page)

```
  +- include/
  |  ... headers.hh ...
  +- src/
     ... sources.cc ...
```

Here, `exampleB1.cc` contains `main()` for the application, with `include/` and `src/` containing the implementation class headers and sources respectively. This arrangement of source files is not mandatory when building with CMake, apart from the location of the `CMakeLists.txt` file in the root directory of the application.

The text file `CMakeLists.txt` is the CMake script containing commands which describe how to build the exampleB1 application

```
# (1)
cmake_minimum_required(VERSION 3.16...3.21)
project(B1)

# (2)
option(WITH_GEANT4_UIVIS "Build example with Geant4 UI and Vis drivers" ON)
if(WITH_GEANT4_UIVIS)
  find_package(Geant4 REQUIRED ui_all vis_all)
else()
  find_package(Geant4 REQUIRED)
endif()

# (3)
include(${Geant4_USE_FILE})
include_directories(${PROJECT_SOURCE_DIR}/include)

# (4)
file(GLOB sources ${PROJECT_SOURCE_DIR}/src/*.cc)
file(GLOB headers ${PROJECT_SOURCE_DIR}/include/*.hh)

# (5)
add_executable(exampleB1 exampleB1.cc ${sources} ${headers})
target_link_libraries(exampleB1 ${Geant4_LIBRARIES})

# (6)
set(EXAMPLEB1_SCRIPTS
  exampleB1.in
  exampleB1.out
  init_vis.mac
  run1.mac
  run2.mac
  vis.mac
  )

foreach(_script ${EXAMPLEB1_SCRIPTS})
  configure_file(
    ${PROJECT_SOURCE_DIR}/${_script}
    ${PROJECT_BINARY_DIR}/${_script}
    COPYONLY
    )
endforeach()

# (7)
install(TARGETS exampleB1 DESTINATION bin)
```

For clarity, the above listing has stripped out the main comments (CMake comments begin with a “#”) you’ll find in the actual file to highlight each distinct task:

1. Basic Configuration
   The `cmake_minimum_required` command and `if` block simply ensures we’re using a suitable version of CMake and that it has been setup appropriately. The `project` command sets the name of the project and enables and configures C and C++ compilers.

2. Find and Configure Geant4
   The aforementioned find_package command is used to locate and configure Geant4 (we'll see how to specify the location later when we run CMake), the REQUIRED argument being supplied so that CMake will fail with an error if it cannot find Geant4. The option command specifies a boolean variable which defaults to ON, and which can be set when running CMake via a -D command line argument, or toggled in the CMake GUI interfaces. We wrap the calls to find_package in a conditional block on the option value. This allows us to configure the use of Geant4 UI and Visualization drivers by exampleB1 via the ui_all vis_all "component" arguments to find_package. An overview of available components is provided *Use of Geant4Config.cmake with find_package in CMake* with a full listing at the top of the installed Geant4Config.cmake file.
3. Configure the Project to Use Geant4 and B1 Headers
   To automatically configure CMake to use additional CMake modules supplied by the Geant4 examples, we use the include command to load a script supplied by Geant4. The CMake variable named Geant4_USE_FILE is set to the path to this module when Geant4 is located by find_package. We use the include_directories command to add the B1 header directory to the compiler's header search path. The CMake variable PROJECT_SOURCE_DIR points to the top level directory of the project and is set by the earlier call to the project command.
4. List the Sources to Build the Application
   Use the globbing functionality of the file command to prepare lists of the B1 source and header files.
   **Note however that CMake globbing is only used here as a convenience**. The expansion of the glob *only happens when CMake is run*, so if you later add or remove files, the generated build scripts will not know a change has taken place. **Kitware strongly recommend listing sources explicitly as CMake automatically makes the build depend on the CMakeLists.txt file**. This means that if you explicitly list the sources in CMakeLists.txt, any changes you make will be automatically picked up when you rebuild. This is also useful when you are working on a project with sources under version control and multiple contributors to ensure traceability and consistent builds.
5. Define and Link the Executable
   The add_executable command defines the build of an application, outputting an executable named by its first argument, with the sources following. Note that we add the headers to the list of sources so that they will appear in IDEs like Xcode.
   After adding the executable, we use the target_link_libraries command to link it with the Geant4 libraries. The Geant4_LIBRARIES variable is set by find_package when Geant4 is located, and is a list of all the libraries needed to link against to use Geant4.
6. Copy any Runtime Scripts to the Build Directory
   Because we want to support out of source builds so that we won't mix CMake generated files with our actual sources, we copy any scripts used by the B1 application to the build directory. We use foreach to loop over the list of scripts we constructed, and configure_file to perform the actual copy.
   Here, the CMake variable PROJECT_BINARY_DIR is set by the earlier call to the project command and points to the directory where we run CMake to configure the build.
7. If Required, Install the Executable
   Use the install command to create an install target that will install the executable to a bin directory under CMAKE_INSTALL_PREFIX.
   If you don't intend your application to be installable, i.e. you only want to use it locally when built, you can leave this out.

This sequence of commands is the most basic needed to compile and link an application with Geant4, and is easily extendable to more involved use cases such as platform specific configuration or using other third party packages (via find_package).

With the CMake script in place, using it to build an application is a two step process. First CMake is run to generate buildscripts to describe the build. By default, these will be Makefiles on Unix platforms, and Visual Studio solutions on Windows, but you can generate scripts for other tools like Xcode and Eclipse if you wish. Second, the buildscripts are run by the chosen build tool to compile and link the application.

A key concept with CMake is that we generate the buildscripts and run the build in a separate directory, the so-called *build directory*, from the directory in which the sources reside, the so-called *source directory*. This is the exact same technique we used when building Geant4 itself. Whilst this may seem awkward to begin with, it is a very useful technique to employ. It prevents mixing of CMake generated files with those of your application, and allows you to have multiple builds against a single source without having to clean up, reconfigure and rebuild.

We'll illustrate this configure and build process on Linux/macOS using Makefiles, and on Windows using Visual Studio. The example script and Geant4's `Geant4Config.cmake` script are vanilla CMake, so you should be able to use other Generators (such as Xcode and Eclipse) without issue.

### Building ExampleB1 with CMake on Unix with Makefiles

We'll assume, *for illustration only*, that you've copied the exampleB1 sources into a directory under your home area so that we have:

```
+- /home/you/B1/
   +- CMakeLists.txt
   +- exampleB1.cc
   +- include/
   +- src/
   +- ...
```

Here, our *source directory* is `/home/you/B1`, in other words the directory holding the `CMakeLists.txt` file.

Let's also assume that you have already installed Geant4 in your home area under, *for illustration only*, `/home/you/geant4-install`.

Our first step is to create a *build directory* in which build the example. We will create this alongside our B1 *source directory* as follows:

```
$ cd $HOME
$ mkdir B1-build
```

We now change to this *build directory* and run CMake to generate the Makefiles needed to build the B1 application. We pass CMake two arguments

```
$ cd $HOME/B1-build
$ cmake -DCMAKE_PREFIX_PATH=/home/you/geant4-install $HOME/B1
```

Here, the first argument points CMake to the install prefix of Geant4. `CMAKE_INSTALL_PREFIX` may be extended with additional paths to search for packages, and also set in the environment. See the CMake documentation on CMAKE_PREFIX_PATH and find_package for more details.

For an exact search, you may also use the `Geant4_DIR` variable, e.g:

```
$ cd $HOME/B1-build
$ cmake -DGeant4_DIR=/home/you/geant4-install/lib/cmake/Geant4 $HOME/B1
```

This variable should set to the directory holding the `Geant4Config.cmake` file for the install of Geant4 you want to use.

The second argument to CMake is the path to the *source directory* of the application we want to build. Here it's just the B1 directory as discussed earlier. You should of course adapt the value of that variable to where you copied the B1 source directory.

CMake will now run to configure the build and generate Makefiles and you will see output similar to

```
$ cmake -DCMAKE_PREFIX_PATH=/home/you/geant4-install $HOME/B1
-- The C compiler identification is GNU 9.3.0
-- The CXX compiler identification is GNU 9.3.0
-- Check for working C compiler: /usr/bin/gcc-9
-- Check for working C compiler: /usr/bin/gcc-9 -- works
```

(continues on next page)

(continued from previous page)

```
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/g++-9
-- Check for working CXX compiler: /usr/bin/g++-9 -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/you/B1-build
```

The exact output will depend on the UNIX variant, compiler, and CMake version but the last three lines should be identical to within the exact path used.

If you now list the contents of you build directory, you can see the files generated:

```
$ ls
CMakeCache.txt       exampleB1.in   Makefile      vis.mac
CMakeFiles           exampleB1.out  run1.mac
cmake_install.cmake  init_vis.mac   run2.mac
```

Note the `Makefile` and that all the scripts for running the exampleB1 application we're about to build have been copied across. With the Makefile available, we can now build by simply running make:

```
$ make -jN
```

CMake generated Makefiles support parallel builds, so `N` can be set to the number of cores on your machine (e.g. on a dual core processor, you could set N to 2). When make runs, you should see the output:

```
$ make
Scanning dependencies of target exampleB1
[ 12%] Building CXX object B1/CMakeFiles/exampleB1.dir/exampleB1.cc.o
[ 25%] Building CXX object B1/CMakeFiles/exampleB1.dir/src/ActionInitialization.cc.o
[ 37%] Building CXX object B1/CMakeFiles/exampleB1.dir/src/DetectorConstruction.cc.o
[ 50%] Building CXX object B1/CMakeFiles/exampleB1.dir/src/EventAction.cc.o
[ 62%] Building CXX object B1/CMakeFiles/exampleB1.dir/src/PrimaryGeneratorAction.cc.o
[ 75%] Building CXX object B1/CMakeFiles/exampleB1.dir/src/RunAction.cc.o
[ 87%] Building CXX object B1/CMakeFiles/exampleB1.dir/src/SteppingAction.cc.o
[100%] Linking CXX executable exampleB1
[100%] Built target exampleB1
```

CMake Unix Makefiles are quite terse, but you can make them more verbose by adding the `VERBOSE` argument to make:

```
$ make VERBOSE=1
```

If you now list the contents of your *build directory* you will see the exampleB1 application executable has been created:

```
$ ls
CMakeCache.txt       exampleB1      init_vis.mac      run2.mac
CMakeFiles           exampleB1.in   Makefile          vis.mac
cmake_install.cmake  exampleB1.out  run1.mac
```

You can now run the application in place:

```
$ ./exampleB1
Available UI session types: [ GAG, tcsh, csh ]

```

(continues on next page)

(continued from previous page)

```
*************************************************************
 Geant4 version Name: geant4-11-02 [MT]   (8-December-2023)
  << in Multi-threaded mode >>
                      Copyright : Geant4 Collaboration
                     References : NIM A 506 (2003), 250-303
                                : IEEE-TNS 53 (2006), 270-278
                                : NIM A 835 (2016), 186-225
                            WWW : http://geant4.org/
*************************************************************

<<< Reference Physics List QBBC
Visualization Manager instantiating with verbosity "warnings (3)"...
Visualization Manager initialising...
Registering graphics systems...
```

Note that the exact output shown will depend on how both Geant4 and your application were configured. Further output and behaviour beyond the `Registering graphics systems...` line will depend on what UI and Visualization drivers your Geant4 install supports. If you recall the use of the `ui_all vis_all` in the `find_package` command, this results in all available UI and Visualization drivers being activated in your application. If you didn't want any UI or Visualization, you could rerun CMake in your build directory with arguments:

```
$ cmake -DWITH_GEANT4_UIVIS=OFF .
```

This would switch the `option` we set up to false, and result in `find_package` not activating any UI or Visualization for the application. You can easily adapt this pattern to provide options for your application such as additional components or features.

Once the build is configured, you can edit code for the application in its *source directory*. You only need to rerun `make` in the corresponding *build directory* to pick up and compile the changes. However, note that due to the use of CMake globbing to create the source file list, if you add or remove files, you must remember to rerun CMake to pick up the changes. This is another reason why Kitware recommend listing the sources explicitly.

### Building ExampleB1 with CMake on Windows with Visual Studio

As with building Geant4 itself, the simplest system to use for building applications on Windows is a Visual Studio Developer Command Prompt, which can be started from *Start* → *Visual Studio 2017* → *Developer Command Prompt for VS2017* (similarly for VS2015)

We'll assume, *for illustration only*, that you've copied the exampleB1 sources into a directory `C:\Users\YourUsername\B1` so that we have:

```
+- C:\Users\YourUsername\B1
   +- CMakeLists.txt
   +- exampleB1.cc
   +- include\
   +- src\
   +- ...
```

Here, our *source directory* is `C:\Users\YourUsername\B1`, in other words the directory holding the `CMakeLists.txt` file.

Let's also assume that you have already installed Geant4 in your home area under, *for illustration only*, `C:\Users\YourUsername\Geant4-install`.

Our first step is to create a *build directory* in which build the example. We will create this alongside our B1 *source directory* as follows, working from the Visual Studio Developer Command Prompt:

```
> cd %HOMEPATH%
> mkdir B1-build
```

We now change to this *build directory* and run CMake to generate the Visual Studio solution needed to build the B1 application. We pass CMake two arguments

```
> cd %HOMEPATH%\Geant4\B1-build
> cmake -DCMAKE_PREFIX_PATH="%HOMEPATH%\Geant4-install" "%HOMEPATH%\B1"
```

Here, the first argument points CMake to the install prefix of Geant4. `CMAKE_INSTALL_PREFIX` may be extended with additional paths to search for packages, and also set in the environment. See the CMake documentation on CMAKE_PREFIX_PATH and find_package for more details. As with the examples above, you can also use the `Geant4_DIR` variable. The second argument is the path to the *source directory* of the application we want to build. Here it's just the B1 directory as discussed earlier. You should of course adapt it to where you copied the B1 source directory. In both cases the arguments are quoted in case of the paths containing spaces.

CMake will now run to configure the build and generate Visual Studio solutions and you will see output similar to

```
-- Building for: Visual Studio 15 2017
-- The C compiler identification is MSVC 19.11.25547.0
-- The CXX compiler identification is MSVC 19.11.25547.0
-- Check for working C compiler: C:/Program Files (x86)/Microsoft Visual Studio/2017/Community/
↪VC/Tools/MSVC/14.11.25503/bin/Hostx86/x86/cl.exe
-- Check for working C compiler: C:/Program Files (x86)/Microsoft Visual Studio/2017/Community/
↪VC/Tools/MSVC/14.11.25503/bin/Hostx86/x86/cl.exe -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: C:/Program Files (x86)/Microsoft Visual Studio/2017/Community/
↪VC/Tools/MSVC/14.11.25503/bin/Hostx86/x86/cl.exe
-- Check for working CXX compiler: C:/Program Files (x86)/Microsoft Visual Studio/2017/Community/
↪VC/Tools/MSVC/14.11.25503/bin/Hostx86/x86/cl.exe -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: C:/Users/YourUsername/B1-build
```

If you now list the contents of you build directory, you can see the files generated:

```
> dir /B
ALL_BUILD.vcxproj
ALL_BUILD.vcxproj.filters
B1.sln
B1.vcxproj
B1.vcxproj.filters
CMakeCache.txt
CMakeFiles
cmake_install.cmake
exampleB1.in
exampleB1.out
exampleB1.vcxproj
exampleB1.vcxproj.filters
init_vis.mac
INSTALL.vcxproj
INSTALL.vcxproj.filters
run1.mac
run2.mac
vis.mac
ZERO_CHECK.vcxproj
ZERO_CHECK.vcxproj.filters
```

Note the `B1.sln` solution file and that all the scripts for running the exampleB1 application we're about to build have been copied across. With the solution available, we can now build by running cmake to drive MSBuild:

```
> cmake --build . --config Release
```

Solution based builds are quite verbose, but you should not see any errors at the end. In the above, we have built the B1 program in `Release` mode, meaning that it is optimized and has no debugging symbols. As with building Geant4 itself, this is chosen to provide optimum performance. If you require debugging information for your application, simply change the argument to `RelWithDebInfo`. Note that in both cases you must match the configuration of your application with that of the Geant4 install, i.e. if you are building the application in `Release` mode, then ensure it uses a `Release` build of Geant4. Link and/or runtime errors may result if mixed configurations are used.

After running the build, if we list the contents of the build directory again we see:

```
> dir /B
ALL_BUILD.vcxproj
ALL_BUILD.vcxproj.filters
B1.sln
B1.vcxproj
B1.vcxproj.filters
CMakeCache.txt
CMakeFiles
cmake_install.cmake
exampleB1.dir
exampleB1.in
exampleB1.out
exampleB1.vcxproj
exampleB1.vcxproj.filters
init_vis.mac
INSTALL.vcxproj
INSTALL.vcxproj.filters
Release
run1.mac
run2.mac
vis.mac
Win32
ZERO_CHECK.vcxproj
ZERO_CHECK.vcxproj.filters

> dir /B Release
exampleB1.exe
...
```

Here, the `Release` subdirectory contains the executable, and the main build directory contains all the `.mac` scripts for running the program. If you build in different modes, the executable for that mode will be in a directory named for that mode, e.g. `RelWithDebInfo/exampleB1.exe`. You can now run the application in place:

```
> .\Release\exampleB1.exe

*************************************************************
 Geant4 version Name: geant4-11-02 [MT]   (8-December-2023)
  << in Multi-threaded mode >>
                      Copyright : Geant4 Collaboration
                     References : NIM A 506 (2003), 250-303
                                : IEEE-TNS 53 (2006), 270-278
                                : NIM A 835 (2016), 186-225
                            WWW : http://geant4.org/
*************************************************************

<<< Reference Physics List QBBC
Visualization Manager instantiating with verbosity "warnings (3)"...
Visualization Manager initialising...
Registering graphics systems...
```

Note that the exact output shown will depend on how both Geant4 and your application were configured. Further output and behaviour beyond the `Registering graphics systems...` line will depend on what UI and Visualization drivers your Geant4 install supports.

Whilst the Visual Studio Developer Command prompt provides the simplest way to build an application, the generated Visual Studio Solution file (`B1.sln` in the above example) may also be opened directly in the Visual Studio IDE. This provides a more comprehensive development and debugging environment, and you should consult its documentation if you wish to use this.

One key CMake related item to note goes back to our listing of the *headers* for the application in the call to `add_executable`. Whilst CMake will naturally ignore these for configuring compilation of the application, it will add them to the Visual Studio Solution. If you do not list them, they will not be editable in the Solution in the Visual Studio IDE.

## 2.8.2 Use of `Geant4Config.cmake` with `find_package` in CMake

The `Geant4Config.cmake` file installed by Geant4 is designed to be used with CMake's find_package command. CMake will search for the file using a standard set of paths used by `find_package`, or via the `Geant4_DIR`. When found, it sets several CMake variables and provides a mechanism for checking and activating optional features of Geant4 if your application requires these. The simplest possible usage of `find_package` and these variables to configure an application or library requiring Geant4 is:

```
find_package(Geant4 REQUIRED)                            # Find Geant4
add_executable(myg4app myg4app.cc)                       # Compile application
target_link_libraries(myg4app ${Geant4_LIBRARIES})  # Link it to Geant4
```

The `Geant4_LIBRARIES` variable holds the list of CMake Imported Targets for the Geant4 libraries. These set and propagate all Usage Requirements of Geant4 to the consuming target(s) (the `myg4app` executable in the above).

The minimal example just requires that a Geant4 install be found. A version number may be supplied to search for an install *greater than or equal to* the supplied version, e.g.

```
find_package(Geant4 10.0 REQUIRED)
```

makes CMake search for a Geant4 install whose version number is greater than or equal to 10.0. An exact version number may also be specified:

```
find_package(Geant4 10.4.0 EXACT REQUIRED)
```

In both cases, CMake will fail with an error if a Geant4 install meeting these version requirements is not found.

Geant4 can be installed with many optional components, and the presence of these can also be required and activated by passing extra "component" arguments. For example, to require that Geant4 is found *and* that it has support for gdml and Qt:

```
find_package(Geant4 REQUIRED gdml qt)
```

which will fail if the found install was not built with these options. If you want to activate components only if they exist, you can use the pattern

```
find_package(Geant4 REQUIRED)
find_package(Geant4 QUIET OPTIONAL_COMPONENTS qt)
```

which will require CMake to locate a core install of Geant4, and then check for and activate Qt support if the install provides it, continuing without error otherwise. A key thing to note here is that you can call `find_package` multiple times to append configuration of components. If you use this pattern and need to check if a component was found, you can use the `Geant4_<COMPONENTNAME>_FOUND` variables which are set after the call to `find_package`.

Some components are "passive" in that they just indicate support is available, others are "active" in that they indicate support for and activate use of the component in the application linking to the targets in `Geant4_LIBRARIES`. A partial list of the most useful components and their behaviour is given below, but for a full list, please see the listing in the installed `Geant4Config.cmake` file.

- `multithreaded`
  `Geant4_multithreaded_FOUND` is `TRUE` if the install of Geant4 was built with multithreading support. Note that this is a **passive** option and only indicates availability of multithreading support! **Multithreading in your application code requires creation and usage of the appropriate C++ objects and interfaces as described in this guide.**
- `gdml`
  `Geant4_gdml_FOUND` is `TRUE` if the install of Geant4 was built with GDML support.
  Note that this is a **passive** option, and indicates support for GDML is availble in the found install.
- `ui_all`
  Activates **all** available UI drivers. Does not set any variables, and never causes CMake to fail. *It is recommended to use this over specific UI drivers unless your application has strong requirements.*
- `vis_all`
  Activates **all** available Visualization drivers. Does not set any variables, and never causes CMake to fail. *It is recommended to use this over specific Vis drivers unless your application has strong requirements.*
- `ui_tcsh`
  `Geant4_ui_tcsh_FOUND` is `TRUE` if the install of Geant4 provides the TCsh command line User Interface. Using this component activates and allows use of the TCsh command line interface in the linked application.
- `ui_win32`
  `Geant4_ui_win32_FOUND` is `TRUE` if the install of Geant4 provides the Win32 command line User Interface. Using this component activates and allows use of the Win32 command line interface in the linked application.
- `motif`
  `Geant4_motif_FOUND` is `TRUE` if the install of Geant4 provides the Motif(Xm) User Interface and Visualization driver. Using this component activates and allows use of the Motif User Interface and Visualization Driver in the linked application.
- `qt`
  `Geant4_qt_FOUND` is `TRUE` if the install of Geant4 provides the Qt User Interface and Visualization driver. Using this component activates and allows use of the Qt User Interface and Visualization Driver in the linked application.
- `vis_raytracer_x11`
  `Geant4_vis_raytracer_x11_FOUND` is `TRUE` if the install of Geant4 provides the X11 interface to the RayTracer Visualization driver. Using this component activates and allows use of the RayTracer X11 Visualization Driver in the linked application.
- `vis_opengl_x11`
  `Geant4_vis_opengl_x11_FOUND` is `TRUE` if the install of Geant4 provides the X11 interface to the OpenGL Visualization driver. Using this component activates and allows use of the X11 OpenGL Visualization Driver in the linked application.
- `vis_opengl_win32`
  `Geant4_vis_opengl_win32_FOUND` is `TRUE` if the install of Geant4 provides the Win32 interface to the OpenGL Visualization driver. Using this component activates and allows use of the Win32 OpenGL Visualization Driver in the linked application.
- `vis_openinventor`
  `Geant4_vis_openinventor_FOUND` is `TRUE` if the install of Geant4 provides the OpenInventor Visualization driver. Using this component activates and allows use of the OpenInventor Visualization Driver in the linked application.
- `vis_toolssg_x11_gles`
  `Geant4_vis_toolssg_x11_gles_FOUND` is `TRUE` if the install of Geant4 provides the ToolsSG visualization driver with X11 backend. Using this component allows use of the ToolsSG Visualization Driver in the linked application.
- `vis_toolssg_xt_gles`
  `Geant4_vis_toolssg_xt_gles_FOUND` is `TRUE` if the install of Geant4 provides the ToolsSG visualization driver with Motif backend. Using this component allows use of the ToolsSG Visualization Driver in the linked application.
- `vis_toolssg_qt_gles`

`Geant4_vis_toolssg_qt_gles_FOUND` is `TRUE` if the install of Geant4 provides the ToolsSG visualization driver with Qt5 backend. Using this component allows use of the ToolsSG Visualization Driver in the linked application.

- `vis_toolssg_windows_gles`
  `Geant4_vis_toolssg_windows_gles_FOUND` is `TRUE` if the install of Geant4 provides the ToolsSG visualization driver with Windows backend. Using this component allows use of the ToolsSG Visualization Driver in the linked application.
- `vis_Vtk`
  `Geant4_vis_Vtk_FOUND` is `TRUE` if the install of Geant4 provides the Vtk visualization driver. Using this component allows use of the Vtk Visualization Driver in the linked application.

## 2.9 How to Set Up an Interactive Session

### 2.9.1 Introduction

#### Roles of the "intercoms" category

The "intercoms" category provides an expandable command interpreter. It is the key mechanism of GEANT4 to realize secure user interactions across categories without being annoyed by dependencies among categories. GEANT4 commands can be used in an interactive session, a batch mode with a macro file, or a direct C++ call.

#### User Interfaces to drive the simulation

GEANT4 can be controlled by a series of GEANT4 UI commands. The "intercoms" category provides the abstract class `G4UIsession` that processes interactive commands. The concrete implementations of (graphical) user interface are provided in the "interfaces" category. The strategy realize to adopt various user interface tools, and allows GEANT4 to utilize the state-of-the-art GUI tools such as Motif, Qt, and Java etc. The following interfaces is currently available;

1. Command-line terminal (dumb terminal and tcsh-like terminal)
2. Xm, Qt, Win32, variations of the above terminal by using a Motif, Qt, Windows widgets
3. GAG, a fully graphical user interface and its network extension GainServer of the client/server type.

Implementation of the user sessions (1 and 2) is included in the `source/interfaces/basic` directory. As for GAG, the front-end class is included in the `source/interfaces/GAG` directory, while its partner GUI package MOMO.jar is available under the `environments/MOMO` directory. MOMO.jar, Java archive file, contains not only GAG, but also GGE and other helper packages.

### 2.9.2 A Short Description of Available Interfaces

#### `G4UIterminal`

This interface opens a session on a command-line terminal. `G4UIterminal` runs on all supported platforms. There are two kinds of shells, `G4UIcsh` and `G4UItcsh`. `G4UItcsh` supports tcsh-like readline features (cursor and command completion) and works on Linux on Mac, while `G4UIcsh` is a plain standard input (cin) shell that works on all platforms. The following built-in commands are available in `G4UIterminal`;

**cd, pwd** change, display the current command directory.
**ls, lc** list commands and subdirectories in the current directory.
**history** show previous commands.
**!historyID** reissue previous command.
**?command** show current parameter values of the command.
**help command** show command help.

**exit** terminate the session.

G4UItcsh supports user-friendly key bindings a-la-tcsh. `G4UItcsh` runs on Linux and Mac. The following keybindings are supported;

**^A** move cursor to the top
**^B** backward cursor ([LEFT] cursor)
**^C (except Windows terminal)** abort a run ( soft abort ) during event processing. A program will be terminated while accepting a user command.
**^D** delete/exit/show matched list
**^E** move cursor to the end
**^F** forward cursor ([RIGHT] cursor)
**^K** clear after the cursor
**^N** next command ([DOWN] cursor)
**^P** previous command ([UP] cursor)
**TAB** command completion
**DEL** backspace
**BS** backspace

The example below shows how to set a user's prompt.

```
G4UItcsh* tcsh = new G4UItcsh();
tcsh-> SetPrompt("%s>");
```

The following strings are supported as substitutions in a prompt string.

**%s** current application status
**%/** current working directory
**%h** history number

Command history in a user's session is saved in a file `$(HOME)/.g4_hist` that is automatically read at the next session, so that command history is available across sessions.

### `G4UIXm`, `G4UIQt` and `G4UIWin32` classes

These interfaces are versions of `G4UIterminal` implemented over libraries Motif, Qt and WIN32 respectively. `G4UIXm` uses the Motif XmCommand widget, `G4UIQt` the Qt dialog widget, and `G4UIWin32` the Windows "edit" component to do the command capturing. These interfaces are useful if working in conjunction with visualization drivers that use the Xt library, Qt library or the WIN32 one.

A command box is at disposal for entering or recalling GEANT4 commands. Command completion by typing "TAB" key is available in the command box. The shell commands "exit, cont, help, ls, cd…" are also supported. A menu bar can be customized through the *AddMenu* and *AddButton* method. Ex:

**/gui/addMenu** test Test
**/gui/addButton** test Init /run/initialize
**/gui/addButton** test "Set gun" "/control/execute gun.g4m"
**/gui/addButton** test "Run one event" "/run/beamOn 1"

`G4UIXm` runs on Unix/Linux with Motif. `G4UIQt` run everywhere with Qt. `G4UIWin32` runs on Windows.

#### `G4UIGAG` and `G4UIGainServer` classes

They are front-end classes of GEANT4 which make connections with their respective graphical user interfaces, GAG (GEANT4 Adaptive GUI) via pipe, and Gain (GEANT4 adaptive interface for network) via sockets. While GAG must run on the same system (Windows or Unixen) as a GEANT4 application, Gain can run on a remote system (Windows, Linux, etc.) in which JRE (Java Runtime Environment) is installed. A GEANT4 application is invoked on a Unix (Linux) system and behaves as a network server. It opens a port, waiting the connection from the Gain. Gain has capability to connect to multiple GEANT4 "servers" on Unixen systems at different hosts.

Client GUIs, GAG and Gain have almost similar look-and-feel. So, GAG's functionalities are briefly explained here. Please refer to the URL previously mentioned for details.

Using GAG, user can select a command, set its parameters and execute it. It is adaptive, in the sense that it reflects the internal states of GEANT4 that is a state machine. So, GAG always provides users with the GEANT4 commands which may be added, deleted, enabled or disabled during a session. GAG does nothing by itself but to play an intermediate between user and an executable simulation program via pipes. GEANT4's front-end class `G4UIGAG` must be instantiated to communicate with GAG. GAG runs on Linux and Windows. MOMO.jar is supplied in the GEANT4 source distribution and can be run by a command:

```
%java -jar  /path/to/geant4.10.00/environments/MOMO/MOMO.jar
```

GAG has following functions.

**GAG Menu:** The menus are to choose and run a GEANT4 executable file, to kill or exit a GEANT4 process and to exit GAG. Upon the normal exit or an unexpected death of the GEANT4 process, GAG window are automatically reset to run another GEANT4 executable.

**GEANT4 Command tree:** Upon the establishment of the pipe connection with the GEANT4 process, GAG displays the command menu, using expandable tree browser whose look and feel is similar to a file browser. Disabled commands are shown in opaque. GAG doesn't display commands that are just below the root of the command hierarchy. Direct type-in field is available for such input. Guidance of command categories and commands are displayed upon focusing. GAG has a command history function. User can re-execute a command with old parameters, edit the history, or save the history to create a macro file.

**Command Parameter panel:** GAG's parameter panel is the user-friendliest part. It displays parameter name, its guidance, its type(s) (integer, double, Boolean or string), omittable, default value(s), expression(s) of its range and candidate list(s) (for example, of units). Range check is done by intercoms and the error message from it is shown in the pop-up dialog box. When a parameter component has a candidate list, a list box is automatically displayed . When a file is requested by a command, the file chooser is available.

**Logging:** Log can be redirected to the terminal (xterm or cygwin window) from which GAG is invoked. It can be interrupted as will, in the middle of a long session of execution. Log can be saved to a file independent of the above redirection . GAG displays warning or error messages from GEANT4 in a pop-up warning widget.

### 2.9.3 How to Select Interface in Your Applications

To choose an interface (`G4UIxxx` where `xxx = terminal,Xm, Win32, Qt, GAG, GainServer`) in your programs, there are two ways.

- Calling G4UIxxx directly:

```
#include "G4Uixxx.hh"

G4UIsession* session = new G4UIxxx;
session-> SessionStart();

delete session;
```

**Note:** For using a tcsh session, `G4UIterminal` is instantiated like:

```
G4UIsession* session = new G4UIterminal(new G4UItcsh);
```

If the user wants to deactivate the default signal handler (soft abort) raised by "Ctr-C", the false flag can be set in the second argument of the `G4UIterminal` constructor like

```
G4UIsession* session = new G4UIterminal(new G4UItcsh, false).
```

- Using `G4UIExecutive` This is more convenient way for choosing a session type, that can select a session at run-time according to a rule described below.

```
#include "G4UIExecutive.hh"

G4UIExecutive* ui = new G4UIExecutive(argc, argv);
ui->SessionStart();

delete ui;
```

`G4UIExecutive` has several ways to choose a session type. A session is selected in the following rule. Note that session types are identified by a case-insensitive characters ("qt", "xm", "win32", "gag", "tcsh", "csh").

1. Check the argument of the constructor of G4UIExecutive. You can specify a session like `new G4UIExecutive(argc, argv, "qt");`
2. Check environment variables, `G4UI_USE_XX (XX= QT, XM, WIN32, GAG, TCSH)`. Select a session if the corresponding environment variable is defined. Variables are checked in the order of QT, XM, WIN32, GAG, TCSH if multiple variables are set.
3. Check `~/.g4session`. You can specify the default session type and a session type by each application in that file. The below shows a sample of `.g4session`.

```
tcsh  # default session
exampleN03 Qt # (application name / session type)
myapp  tcsh
hoge csh
```

4. Guess the best session type according to build session libraries. The order of the selection is Qt, tcsh, Xm.

In any cases, `G4UIExecutive` checks if a specified session is build or not. If not, it goes the next step. A terminal session with csh is the fallback session. If none of specified session is available, then it will be selected.

## 2.10 How to Execute a Program

### 2.10.1 Introduction

A GEANT4 application can be run either in

- 'purely hard-coded' batch mode
- batch mode, but reading a macro of commands
- interactive mode, driven by command lines
- interactive mode via a Graphical User Interface

The last mode will be covered in *How to Set Up an Interactive Session*. The first three modes are explained here.

## 2.10.2 'Hard-coded' Batch Mode

Below is a modified main program of the basic example B1 to represent an application which will run in batch mode.

Listing 2.19: An example of the `main()` routine for an application which will run in batch mode.

```
using namespace B1;

int main()
{
  // Construct the default run manager
  auto runManager = G4RunManagerFactory::CreateRunManager();

  // Set mandatory initialization classes
  runManager->SetUserInitialization(new DetectorConstruction);
  runManager->SetUserInitialization(new QGSP_BIC_EMY);
  runManager->SetUserInitialization(new ActionInitialization);

  // Initialize G4 kernel
  runManager->Initialize();

  // start a run
  int numberOfEvent = 1000;
  runManager->BeamOn(numberOfEvent);

  // job termination
  delete runManager;
  return 0;
}
```

Even the number of events in the run is 'frozen'. To change this number you must at least recompile `main()`.

## 2.10.3 Batch Mode with Macro File

Below is a modified main program of the basic example B1 to represent an application which will run in batch mode, but reading a file of commands.

Listing 2.20: An example of the `main()` routine for an application which will run in batch mode, but reading a file of commands.

```
using namespace B1;

int main(int argc,char** argv)
{
  // Construct the default run manager
  auto runManager = G4RunManagerFactory::CreateRunManager();

  // Set mandatory initialization classes
  runManager->SetUserInitialization(new DetectorConstruction);
  runManager->SetUserInitialization(new QGSP_BIC_EMY);
  runManager->SetUserInitialization(new ActionInitialization);

  // Initialize G4 kernel
  runManager->Initialize();

  //read a macro file of commands
  G4UImanager* UI = G4UImanager::GetUIpointer();
  G4String command = "/control/execute ";
  G4String fileName = argv[1];
  UI->ApplyCommand(command+fileName);

```

(continues on next page)

(continued from previous page)

```
  // job termination
  delete runManager;
  return 0;
}
```

This example will be executed with the command:

```
> exampleB1  run1.mac
```

where `exampleB1` is the name of the executable and `run1.mac` is a macro of commands located in the current directory, which could look like:

Listing 2.21: A typical command macro.

```
#
# Macro file for myProgram
#
# set verbose level for this run
#
/run/verbose      2
/event/verbose    0
/tracking/verbose 1
#
# Set the initial kinematic and run 100 events
# electron 1 GeV to the direction (1.,0.,0.)
#
/gun/particle e-
/gun/energy 1 GeV
/run/beamOn 100
```

Indeed, you can re-execute your program with different run conditions without recompiling anything.

**Note:** many G4 category of classes have a verbose flag which controls the level of 'verbosity'.

Usually `verbose=0` means silent. For instance

- `/run/verbose` is for the `RunManager`
- `/event/verbose` is for the `EventManager`
- `/tracking/verbose` is for the `TrackingManager`
- …etc…

### 2.10.4 Interactive Mode Driven by Command Lines

Below is an example of the main program for an application which will run interactively, waiting for command lines entered from the keyboard.

Listing 2.22: An example of the `main()` routine for an application which will run interactively, waiting for commands from the keyboard.

```
using namespace B1;

int main(int argc,char** argv)
{
  // Construct the default run manager
  G4RunManager* runManager = new G4RunManager;

  // Set mandatory initialization classes
```

(continues on next page)

(continued from previous page)

```
  runManager->SetUserInitialization(new DetectorConstruction);
  runManager->SetUserInitialization(new QGSP_BIC_EMY);
  runManager->SetUserInitialization(new ActionInitialization);

  // Initialize G4 kernel
  runManager->Initialize();

  // Define UI terminal for interactive mode
  G4UIsession * session = new G4UIterminal;
  session->SessionStart();
  delete session;

  // job termination
  delete runManager;
  return 0;
}
```

This example will be executed with the command:

```
> exampleB1
```

where `exampleB1` is the name of the executable.

The G4 kernel will prompt:

```
Idle>
```

and you can start your session. An example session could be:

Run 5 events:

```
Idle> /run/beamOn 5
```

Switch on tracking/verbose and run one more event:

```
Idle> /tracking/verbose 1
Idle> /run/beamOn 1
```

Change primary particle type an run more events:

```
Idle> /gun/particle mu+
Idle> /gun/energy 10 GeV
Idle> /run/beamOn 1
Idle> /gun/particle proton
Idle> /gun/energy 100 MeV
Idle> /run/beamOn 3
Idle> exit
```

For the meaning of the machine state `Idle`, see *as a state machine*.

This mode is useful for running a few events in debug mode and visualizing them. How to include visualization will be shown in the next, general case, example.

## 2.10.5 General Case

All basic examples in the `examples/basic` subdirectory of the GEANT4 source distribution have the following `main()` structure. The application can be run either in batch or interactive mode.

Listing 2.23: The typical `main()` routine from the examples directory.

```
using namespace B1;

int main(int argc,char** argv)
{
  // Detect interactive mode (if no arguments) and define UI session
  G4UIExecutive* ui = 0;
  if ( argc == 1 ) {
    ui = new G4UIExecutive(argc, argv);
  }

  // Optionally: choose a different Random engine...
  // G4Random::setTheEngine(new CLHEP::MTwistEngine);

  // Construct the default run manager
  G4RunManager* runManager = new G4RunManager;

  // Set mandatory initialization classes
  //
  // Detector construction
  runManager->SetUserInitialization(new DetectorConstruction());

  // Physics list
  G4VModularPhysicsList* physicsList = new QBBC;
  physicsList->SetVerboseLevel(1);
  runManager->SetUserInitialization(physicsList);

  // User action initialization
  runManager->SetUserInitialization(new ActionInitialization());

  // Initialize visualization
  G4VisManager* visManager = new G4VisExecutive;
  // G4VisExecutive can take a verbosity argument - see /vis/verbose guidance.
  // G4VisManager* visManager = new G4VisExecutive("Quiet");
  visManager->Initialize();

  // Get the pointer to the User Interface manager
  G4UImanager* UImanager = G4UImanager::GetUIpointer();

  // Process macro or start UI session
  if ( ! ui ) {
    // batch mode
    G4String command = "/control/execute ";
    G4String fileName = argv[1];
    UImanager->ApplyCommand(command+fileName);
  } else {
    // interactive mode
    UImanager->ApplyCommand("/control/execute init_vis.mac");
    ui->SessionStart();
    delete ui;
  }

  // Job termination
  // Free the store: user actions, physics_list and detector_description are
  // owned and deleted by the run manager, so they should not be deleted
  // in the main() program !

  delete visManager;
  delete runManager;
}
```

Listing 2.24: The `init.mac` macro

```
# Macro file for the initialization phase of example B1
# when running in interactive mode without visualization
#
# Set some default verbose
/control/verbose 2
/control/saveHistory
/run/verbose 2
```

The `init_vis.mac` macro has just added a line with a call to vis.mac:

```
# Macro file for the initialization phase of example B1
# when running in interactive mode with visualization
#
# Set some default verbose
#
/control/verbose 2
/control/saveHistory
/run/verbose 2
#
# Visualization setting
/control/execute vis.mac
```

The `vis.mac` macro defines a minimal setting for drawing volumes and trajectories accumulated for all events of a given run:

```
# Macro file for the visualization setting in the initialization phase
# of the B1 example when running in interactive mode
#
#
# Use this open statement to create an OpenGL view:
/vis/open OGL 600x600-0+0
#
# Draw geometry:
/vis/drawVolume
#
# Specify view angle:
/vis/viewer/set/viewpointThetaPhi 90. 180.
#
# Draw smooth trajectories at end of event, showing trajectory points
# as markers 2 pixels wide:
/vis/scene/add/trajectories smooth
#
# To superimpose all of the events from a given run:
/vis/scene/endOfEventAction accumulate
#
# Re-establish auto refreshing and verbosity:
/vis/viewer/set/autoRefresh true
/vis/verbose warnings
#
# For file-based drivers, use this to create an empty detector view:
#/vis/viewer/flush
```

Also, this example demonstrates that you can read and execute a macro from another macro or interactively:

```
Idle> /control/execute  mySubMacro.mac
```

## 2.11 How to Visualize the Detector and Events

### 2.11.1 Introduction

This section briefly explains how to perform GEANT4 Visualization. The description here is based on the sample program `examples/basic/B1`. More details are given in *Visualization*.

### 2.11.2 Visualization Drivers

The GEANT4 visualization system was developed in response to a diverse set of requirements:

1. Quick response to study geometries, trajectories and hits
2. High-quality output for publications
3. Flexible camera control to debug complex geometries
4. Tools to show volume overlap errors in detector geometries
5. Interactive picking to get more information on visualized objects

No one graphics system is ideal for all of these requirements, and many of the large software frameworks into which GEANT4 has been incorporated already have their own visualization systems, so GEANT4 visualization was designed around an abstract interface that supports a diverse family of graphics systems. Some of these graphics systems use a graphics library compiled with GEANT4, such as OpenGL, Qt or OpenInventor, while others involve a separate application, such as HepRApp or DAWN.

You need not use all visualization drivers. You can select those suitable to your purposes. In the following, for simplicity, we assume that the GEANT4 libraries are built with the Qt driver.

If you build GEANT4 using the standard `CMake` procedure, you include Qt by setting GEANT4_USE_QT to ON.

In order to use the the Qt driver, you need the OpenGL library, which is installed in many platforms by default and `CMake` will find it. (If you wish to "do-it-yourself", see *Installing Visualization Drivers*.) The makefiles then set appropriate C-pre-processor flags to select appropriate code at compilation time.

If you are using multithreaded mode, from GEANT4 version 10.2 event drawing is performed by a separate thread and you may need to optimise this with special `/vis/multithreading` commands - see *Multithreading commands*.

### 2.11.3 How to Incorporate Visualization Drivers into an Executable

Most GEANT4 examples already incorporate visualization drivers. If you want to include visualization in your own GEANT4 application, you need to instantiate and initialize a subclass of `G4VisManager` that implements the pure virtual function `RegisterGraphicsSystems()`.

The provided class `G4VisExecutive` can handle all of this work for you. `G4VisExecutive` is sensitive to the `G4VIS_...` variables (that you either set by hand or that are set for you by GNUMake or CMake configuration):

```
auto visManager = new G4VisExecutive(argc, argv);
```

See below for how to use in your `main` program. Basic example B1 is a good place to look..

If you really want to write your own subclass, rather than use `G4VisExecutive`, you may do so. You will see how to do this by looking at `G4VisExecutive.icc`. This subclass must be compiled in the user's domain to force the loading of appropriate libraries in the right order. A typical extract is:

```
...
  RegisterGraphicsSystem (new G4DAWNFILE);
...
#ifdef G4VIS_USE_OPENGLX
```

(continues on next page)

(continued from previous page)

```
  RegisterGraphicsSystem (new G4OpenGLImmediateX);
  RegisterGraphicsSystem (new G4OpenGLStoredX);
#endif
...
```

The `G4VisExecutive` takes ownership of all registered graphics systems, and will delete them when it is deleted at the end of the user's job (see below).

If you wish to use `G4VisExecutive` but register an additional graphics system, `XXX` say, you may do so either before or after initializing:

```
visManager->RegisterGraphicsSytem(new XXX);
visManager->Initialize();
```

An example of a typical `main()` function is given below.

## 2.11.4 Writing the `main()` Method to Include Visualization

Now we explain how to write a visualization manager and the `main()` function for GEANT4 visualization. In order that your GEANT4 executable is able to perform visualization, you must instantiate and initialize *your* Visualization Manager in the `main()` function. The typical `main()` function available for visualization is written in the following style:

Listing 2.25: The typical `main()` routine available for visualization.

```
.....
#include "G4VisExecutive.hh"
.....
int main(int argc,char** argv) {
  .....
  // Initialize visualization with the default graphics system
  auto visManager = new G4VisExecutive(argc, argv);
  // Constructors can also take optional arguments:
  // - a graphics system of choice, eg. "OGL"
  // - and a verbosity argument - see /vis/verbose guidance.
  // auto visManager = new G4VisExecutive(argc, argv, "OGL", "Quiet");
  // auto visManager = new G4VisExecutive("Quiet");
  visManager->Initialize();
  .....
  // Job termination
  delete visManager;
  .....
  return 0;
}
```

We recommend you choose the graphics driver at run time - see *Controlling Visualization from Commands*. This gives you flexibility to switch drivers easily.

Note that we are here recommending that all jobs instantiate a Visualization Manager. Even in batch mode you may generate an image using one of the file-writing drivers - TSG_OFFSCREEN, VTK_OFFSCREEN, DAWNFILE, VRML2FILE, HepRepFile, RayTracer.

Note also that it is your responsibility to delete the Visualization Manager. A good example of a `main()` function is `examples/basic/B1/exampleB1.cc`.

### 2.11.5 Sample Visualization Sessions

Most GEANT4 examples include a `vis.mac`. Run that macro to see a typical visualization. Read the comments in the macro to learn a little bit about some visualization commands. The `vis.mac` also includes commented-out optional visualization commands. By uncommenting some of these you can see additional visualization features.

### 2.11.6 For More Information on GEANT4 Visualization

See the *Visualization* part of this user guide.

CHAPTER

THREE

# TOOLKIT FUNDAMENTALS

## 3.1 Class Categories and Domains

### 3.1.1 What is a class category?

In the design of a large software system such as GEANT4, it is essential to partition it into smaller logical units. This makes the design well organized and easier to develop. Once the logical units are defined independent to each other as much as possible, they can be developed in parallel without serious interference.

In object-oriented analysis and design methodology by Grady Booch [Booch1994], class categories are used to create logical units. They are defined as "clusters of classes that are themselves cohesive, but are loosely coupled relative to other clusters." This means that a class category contains classes which have a close relationship (for example, the "has-a" relation). However, relationships between classes which belong to different class categories are weak, i.e., only limited classes of these have "uses" relations. The class categories and their relations are presented by a class category diagram. The class category diagram designed for GEANT4 is shown in the figure below (Fig. 3.1). Each box in the figure represents a class category, and a "uses" relation by a straight line. The circle at an end of a straight line means the class category which has this circle uses the other category.

The file organization of the GEANT4 codes follows basically the structure of this class category. This *User's Manual* is also organized according to class categories.

In the development and maintenance of GEANT4, one software team will be assigned to a class category. This team will have a responsibility to develop and maintain all classes belonging to the class category.

### 3.1.2 Class categories in GEANT4

The following is a brief summary of the role of each class category in GEANT4.

1. **Run and Event**
   These are categories related to the generation of events, interfaces to event generators, and any secondary particles produced. Their roles are principally to provide particles to be tracked to the Tracking Management.
2. **Tracking and Track**
   These are categories related to propagating a particle by analyzing the factors limiting the step and applying the relevant physics processes. The important aspect of the design was that a generalized GEANT4 physics process (or interaction) could perform actions, along a tracking step, either localized in space, or in time, or distributed in space and time (and all the possible combinations that could be built from these cases).
3. **Geometry and Magnetic Field**
   These categories manage the geometrical definition of a detector (solid modeling) and the computation of distances to solids (also in a magnetic field). The GEANT4 geometry solid modeler is based on the ISO STEP standard and it is fully compliant with it. A key feature of the GEANT4 geometry is that the volume definitions are independent of the solid representation. By this abstract interface for the G4 solids, the tracking component works identically for various representations. The treatment of the propagation in the presence of fields

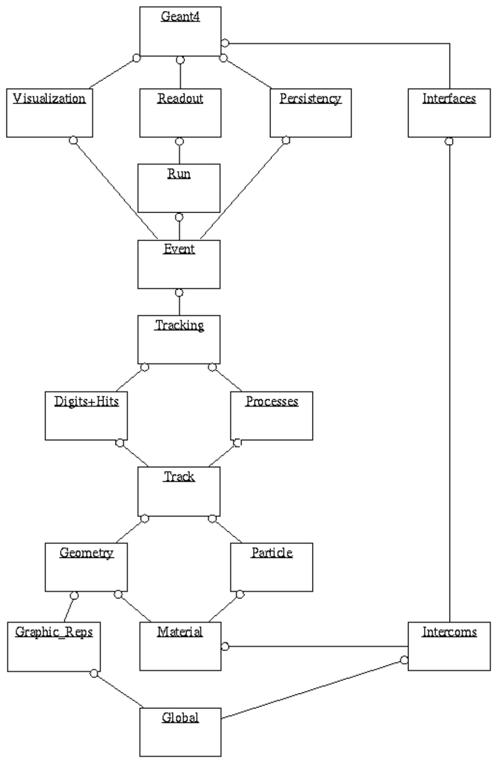


Fig. 3.1: Class categories in GEANT4.

has been provided within specified accuracy. An OO design allows to exchange different numerical algorithms and/or different fields (not only B-field), without affecting any other component of the toolkit.

4. **Particle Definition and Matter**
   These two categories manage the the definition of materials and particles.
5. **Physics**
   This category manages all physics processes participating in the interactions of particles in matter. The abstract interface of physics processes allows multiple implementations of physics models per interaction or per channel. Models can be selected by energy range, particle type, material, etc. Data encapsulation and polymorphism make it possible to give transparent access to the cross sections (independently of the choice of reading from an ascii file, or of interpolating from a tabulated set, or of computing analytically from a formula). Electromagnetic and hadronic physics were handled in a uniform way in such a design, opening up the physics to the users.
6. **Hits and Digitization**
   These two categories manage the creation of hits and their use for the digitization phase. The basic design and implementation of the Hits and Digi had been realized, and also several prototypes, test cases and scenarios had been developed before the alpha-release. Volumes (not necessarily the ones used by the tracking) are aggregated in sensitive detectors, while hits collections represent the logical read out of the detector. Different ways of creating and managing hits collections had been delivered and tested, notably for both single hits and calorimetry hits types. In all cases, hits collections had been successfully stored into and retrieved from an Object Data Base Management System.
7. **Visualization**
   This manages the visualization of solids, trajectories and hits, and interacts with underlying graphical libraries (the Visualization class category). The basic and most frequently used graphics functionality had been implemented already by the alpha-release. The OO design of the visualization component allowed us to develop several drivers independently, such as for OpenGL, Qt and OpenInventor (for X11 and Windows), DAWN, Postscript (via DAWN) and VRML.

8. **Interfaces**
   This category handles the production of the graphical user interface (GUI) and the interactions with external software (OODBMS, reconstruction etc.).

## 3.2 Global Usage Classes

The "global" category in GEANT4 collects all classes, types, structures and constants which are considered of general use within the GEANT4 toolkit. This category also defines the interface with third-party software libraries (CLHEP, STL, etc.) and system-related types, by defining, where appropriate, `typedefs` according to the GEANT4 code conventions.

### 3.2.1 Signature of GEANT4 classes

In order to keep an homogeneous naming style, and according to the GEANT4 coding style conventions, each class part of the GEANT4 kernel has its name beginning with the prefix `G4`, e.g., `G4VHit, G4GeometryManager, G4ProcessVector`, etc. Instead of the raw C types, `G4` types are used within the GEANT4 code. For the basic numeric types (`int, float, double`, etc.), different compilers and different platforms provide different value ranges. In order to assure portability, the use of `G4int, G4float, G4double, G4bool`, globally defined, is preferable. `G4` types implement the right generic type for a given architecture.

#### Basic types

The basic types in GEANT4 are considered to be the following:

- `G4int`,
- `G4long`,
- `G4float`,
- `G4double`,
- `G4bool`,
- `G4complex`,
- `G4String`.

which currently consist of simple `typedefs` to respective types defined in the **CLHEP**, **STL** or system libraries. Most definitions of these basic types come with the inclusion of a single header file, `globals.hh`. This file also provides inclusion of required system headers, as well as some global utility functions needed and used within the GEANT4 kernel.

#### Typedefs to CLHEP classes and their usage

The following classes are `typedefs` to the corresponding classes of the **CLHEP** (**Computing Library for High Energy Physics**) distribution. For more detailed documentation please refer to the CLHEP documentation.

- `G4ThreeVector, G4RotationMatrix, G4LorentzVector` and `G4LorentzRotation`:
  Vector classes: defining 3-component (x,y,z) vector entities, rotation of such objects as 3x3 matrices, 4-component (x,y,z,t) vector entities and their rotation as 4x4 matrices.
- `G4Plane3D, G4Transform3D, G4Normal3D, G4Point3D, G4Scale3D`, and `G4Vector3D`:
  Geometrical classes: defining geometrical entities and transformations in 3D space.

## 3.2.2 The *HEPRandom* module in CLHEP

The *HEPRandom* module, originally part of the GEANT4 kernel, and now distributed as a module of **CLHEP**, has been designed and developed starting from the *Random* class of MC++, the original **CLHEP**'s *HepRandom* module and the **Rogue Wave** approach in the **Math.h++** package. For detailed documentation on the *HEPRandom* classes see the CLHEP documentation.

Information written in this manual is extracted from the original manifesto distributed with the *HEPRandom* package.

The *HEPRandom* module consists of classes implementing different random `engines` and different random `distributions`. A distribution associated to an engine constitutes a random `generator`. A distribution class can collect different algorithms and different calling sequences for each method to define distribution parameters or range-intervals. An engine implements the basic algorithm for pseudo-random numbers generation.

There are 3 different ways of shooting random values:

1. Using the static generator defined in the *HepRandom* class: random values are shot using static methods `shoot()` defined for each distribution class. The static generator will use, as default engine, a *MixMaxRng* object, and the user can set its properties or change it with a new instantiated engine object by using the static methods defined in the *HepRandom* class.
2. Skipping the static generator and specifying an engine object: random values are shot using static methods `shoot(*HepRandomEngine)` defined for each distribution class. The user must instantiate an engine object and give it as argument to the shoot method. The generator mechanism will then be by-passed by using the basic `flat()` method of the specified engine. The user must take care of the engine objects he/she instantiates.
3. Skipping the static generator and instantiating a distribution object: random values are shot using `fire()` methods (NOT static) defined for each distribution class. The user must instantiate a distribution object giving as argument to the constructor an engine by pointer or by reference. By doing so, the engine will be associated to the distribution object and the generator mechanism will be by-passed by using the basic `flat()` method of that engine.

In this guide, we'll only focus on the static generator (point 1.), since the static interface of *HEPRandom* is the only one used within the GEANT4 toolkit.

### *HEPRandom* engines

The class *HepRandomEngine* is the abstract class defining the interface for each random engine. It implements the `getSeed()` and `getSeeds()` methods which return the `initial seed` value and the initial array of seeds (if any) respectively. Many concrete random engines can be defined and added to the structure, simply making them inheriting from *HepRandomEngine*. Several different engines are currently implemented in *HepRandom*, we describe here five of them:

- *HepJamesRandom*
  It implements the algorithm described in *F.James, Comp. Phys. Comm. 60 (1990) 329* for pseudo-random number generation.
- *DRand48Engine*
  Random engine using the `drand48()` and `srand48()` system functions from C standard library to implement the `flat()` basic distribution and for setting seeds respectively. *DRand48Engine* uses the `seed48()` function from C standard library to retrieve the current internal status of the generator, which is represented by 3 short values. *DRand48Engine* is the only engine defined in *HEPRandom* which intrinsically works in 32 bits precision. Copies of an object of this kind are not allowed.
- *MixMaxRng*
  Random number engine implementing the MixMax Matrix Generator of Pseudorandom Numbers generator proposed by *N.Z.Akopov, G.K.Saviddy and N.G.Ter-Arutyunian, J.Compt.Phy. 97, (1991) 573* and *G.Savvidy and N.Savvidy, J.Comput.Phys. 97 (1991) 566*. This is the default random engine for the static generator; it will be invoked by each distribution class unless the user sets a different one.

- *RanluxEngine*
  The algorithm for *RanluxEngine* has been taken from the original implementation in FORTRAN77 by Fred James, part of the **MATHLIB HEP** library. The initialisation is carried out using a Multiplicative Congruential generator using formula constants of L'Ecuyer as described in *F.James, Comp. Phys. Comm. 60 (1990) 329-344*. The engine provides five different luxury levels for quality of random generation. When instantiating a *RanluxEngine*, the user can specify the luxury level to the constructor (if not, the default value 3 is taken). For example:

  ```
  RanluxEngine theRanluxEngine(seed,4);
  // instantiates an engine with `seed' and the best luxury-level
  ... or
  RanluxEngine theRanluxEngine;
  // instantiates an engine with default seed value and luxury-level
  ...
  ```

  The class provides a `getLuxury()` method to get the engine luxury level.
  The `SetSeed()` and `SetSeeds()` methods to set the initial seeds for the engine, can be invoked specifying the luxury level. For example:

  ```
  // static interface
  HepRandom::setTheSeed(seed,4);  // sets the seed to `seed' and luxury to 4
  HepRandom::setTheSeed(seed);    // sets the seed to `seed' keeping
                                  // the current luxury level
  ```

- *RanecuEngine*
  The algorithm for *RanecuEngine* is taken from the one originally written in FORTRAN77 as part of the **MATHLIB HEP** library. The initialisation is carried out using a Multiplicative Congruential generator using formula constants of L'Ecuyer as described in *F.James, Comp. Phys. Comm. 60 (1990) 329-344*. Handling of seeds for this engine is slightly different than the other engines in *HEPRandom*. Seeds are taken from a seed table given an index, the `getSeed()` method returns the current index of seed table. The `setSeeds()` method will set seeds in the local `SeedTable` at a given position index (if the index number specified exceeds the table's size, `[index%size]` is taken). For example:

  ```
  // static interface
  const G4long* table_entry;
  table_entry = HepRandom::getTheSeeds();
  // it returns a pointer `table_entry' to the local SeedTable
  // at the current `index' position. The couple of seeds
  // accessed represents the current `status' of the engine itself !
  ...
  G4int index=n;
  G4long seeds[2];
  HepRandom::setTheSeeds(seeds,index);
  // sets the new `index' for seeds and modify the values inside
  // the local SeedTable at the `index' position. If the index
  // is not specified, the current index in the table is considered.
  ...
  ```

  The `setSeed()` method resets the current `status' of the engine to the original seeds stored in the static table of seeds in *HepRandom*, at the specified index.

Except for the *RanecuEngine*, for which the internal status is represented by just a couple of longs, all the other engines have a much more complex representation of their internal status, which currently can be obtained only through the methods `saveStatus()`, `restoreStatus()` and `showStatus()`, which can also be statically called from *HepRandom*. The status of the generator is needed for example to be able to reproduce a run or an event in a run at a given stage of the simulation.

*RanecuEngine* is probably the most suitable engine for this kind of operation, since its internal status can be fetched/reset by simply using `getSeeds()`/`setSeeds()` (`getTheSeeds()`/`setTheSeeds()` for the static interface in *HepRandom*).

### The static interface in the *HepRandom* class

*HepRandom* a singleton class and using a *MixMaxRng* engine as default algorithm for pseudo-random number generation. *HepRandom* defines a static private data member, `theGenerator`, and a set of static methods to manipulate it. By means of `theGenerator`, the user can change the underlying engine algorithm, get and set the seeds, and use any kind of defined random distribution. The static methods `setTheSeed()` and `getTheSeed()` will set and get respectively the `initial` seed to the main engine used by the static generator. For example:

```
HepRandom::setTheSeed(seed);  // to change the current seed to 'seed'
int startSeed = HepRandom::getTheSeed();  // to get the current initial seed
HepRandom::saveEngineStatus();    // to save the current engine status on file
HepRandom::restoreEngineStatus(); // to restore the current engine to a previous
                                  // saved configuration
HepRandom::showEngineStatus();    // to display the current engine status to stdout
...
int index=n;
long seeds[2];
HepRandom::getTheTableSeeds(seeds,index);
  // fills `seeds' with the values stored in the global
  // seedTable at position `index'
```

Only one random engine can be active at a time, the user can decide at any time to change it, define a new one (if not done already) and set it. For example:

```
RanecuEngine theNewEngine;
HepRandom::setTheEngine(&theNewEngine);
 ...
```

or simply setting it to an old instantiated engine (the old engine status is kept and the new random sequence will start exactly from the last one previously interrupted). For example:

```
HepRandom::setTheEngine(&myOldEngine);
```

Other static methods defined in this class are:

- `void setTheSeeds(const G4long* seeds, G4int)`
- `const G4long* getTheSeeds()`
  To set/get an array of seeds for the generator, in the case of a *RanecuEngine* this corresponds also to set/get the current status of the engine.
- `HepRandomEngine* getTheEngine()`
  To get a pointer to the current engine used by the static generator.

### *HEPRandom* distributions

A distribution-class can collect different algorithms and different calling sequences for each method to define distribution parameters or range-intervals; it also collects methods to fill arrays, of specified size, of random values, according to the distribution. This class collects either static and not static methods. A set of distribution classes are defined in *HEPRandom*. Here is the description of some of them:

- *RandFlat* Class to shoot flat random values (integers or double) within a specified interval. The class provides also methods to shoot just random bits.
- *RandExponential* Class to shoot exponential distributed random values, given a mean (default mean = 1)
- *RandGauss* Class to shoot Gaussian distributed random values, given a mean (default = 0) or specifying also a deviation (default = 1). Gaussian random numbers are generated two at the time, so every other time a number is shot, the number returned is the one generated the time before.
- *RandBreitWigner* Class to shoot numbers according to the Breit-Wigner distribution algorithms (plain or mean^2).

- *RandPoisson* Class to shoot numbers according to the Poisson distribution, given a mean (default = 1) (Algorithm taken from W.H.Press et al., *Numerical Recipes in C*, Second Edition).

## 3.2.3 The *HEPNumerics* module

A set of classes implementing numerical algorithms has been developed in GEANT4. Most of the algorithms and methods have been implemented mainly based on recommendations given in the books:

- B.H. Flowers, *An introduction to Numerical Methods In C++*, Clarendon Press, Oxford 1995.
- M. Abramowitz, I. Stegun, *Handbook of mathematical functions*, DOVER Publications INC, New York 1965 ; chapters 9, 10, and 22.

This set of classes includes:

- `G4ChebyshevApproximation` Class creating the Chebyshev approximation for a function pointed by fFunction data member. The Chebyshev polynomial approximation provides an efficient evaluation of the minimax polynomial, which (among all polynomials of the same degree) has the smallest maximum deviation from the true function.
- `G4DataInterpolation` Class providing methods for data interpolations and extrapolations: Polynomial, Cubic Spline, . . .
- `G4GaussChebyshevQ`
- `G4GaussHermiteQ`
- `G4GaussJacobiQ`
- `G4GaussLaguerreQ` Classes implementing the Gauss-Chebyshev, Gauss-Hermite, Gauss-Jacobi, Gauss-Laguerre and Gauss-Legendre quadrature methods. Roots of orthogonal polynomials and corresponding weights are calculated based on iteration method (by bisection Newton algorithm).
- `G4Integrator` Template class collecting integrator methods for generic functions (Legendre, Simpson, Adaptive Gauss, Laguerre, Hermite, Jacobi).
- `G4SimpleIntegration` Class implementing simple numerical methods (Trapezoidal, MidPoint, Gauss, Simpson, Adaptive Gauss, for integration of functions with signature: double f(double).

## 3.2.4 General management classes

The `global` category defines also a set of `utility` classes generally used within the kernel of GEANT4. These classes include:

- `G4Allocator`

  A class for fast allocation of objects to the heap through paging mechanism. It's meant to be used by associating it to the object to be allocated and defining for it `new` and `delete` operators via `MallocSingle()` and `FreeSingle()` methods of `G4Allocator`.

  **Note:** `G4Allocator` assumes that objects being allocated have all the same size for the type they represent. For this reason, classes which are handled by `G4Allocator` should *avoid* to be used as base-classes for others. Similarly, base-classes of sub-classes handled through `G4Allocator` should not define their (eventually empty) virtual destructors inlined; such measure is necessary in order also to prevent bad aliasing optimisations by compilers which may potentially lead to crashes in the attempt to free allocated chunks of memory when using the base-class pointer or not.

  The list of allocators implicitly defined and used in GEANT4 is reported here:
  - events (`G4Event`): anEventAllocator
  - tracks (`G4Track`): aTrackAllocator
  - stacked tracks (`G4StackedTrack`): aStackedTrackAllocator
  - primary particles (`G4PrimaryParticle`): aPrimaryParticleAllocator
  - primary vertices (`G4PrimaryVertex`): aPrimaryVertexAllocator

  - decay products (`G4DecayProducts`): aDecayProductsAllocator
  - digits collections of an event (`G4DCofThisEvent`): anDCoTHAllocator
  - digits collections (`G4DigiCollection`): aDCAllocator
  - hits collections of an event (`G4HCofThisEvent`): anHCoTHAllocator
  - hits collections (`G4HitsCollection`): anHCAllocator
  - touchable histories (`G4TouchableHistory`): aTouchableHistoryAllocator
  - trajectories (`G4Trajectory`): aTrajectoryAllocator
  - trajectory points (`G4TrajectoryPoint`): aTrajectoryPointAllocator
  - trajectory containers (`G4TrajectoryContainer`): aTrajectoryContainerAllocator
  - navigation levels (`G4NavigationLevel`): aNavigationLevelAllocator
  - navigation level nodes (`G4NavigationLevelRep`): aNavigLevelRepAllocator
  - reference-counted handles (`G4ReferenceCountedHandle<X>`): aRCHAllocator
  - counted objects (`G4CountedObject<X>`): aCountedObjectAllocator
  - HEPEvt primary particles (`G4HEPEvtParticle`): aHEPEvtParticleAllocator
  - electron occupancy objects(`G4ElectronOccupancy`): aElectronOccupancyAllocator
  - "rich" trajectories (`G4RichTrajectory`): aRichTrajectoryAllocator
  - "rich" trajectory points (`G4RichTrajectoryPoint`): aRichTrajectoryPointAllocator
  - "smooth" trajectories (`G4SmoothTrajectory`): aSmoothTrajectoryAllocator
  - "smooth" trajectory points (`G4SmoothTrajectoryPoint`): aSmoothTrajectoryPointAllocator
  - "ray" trajectories (`G4RayTrajectory`): G4RayTrajectoryAllocator
  - "ray" trajectory points (`G4RayTrajectoryPoint`): G4RayTrajectoryPointAllocator

  For each of these allocators, accessible from the global namespace, it is possible to monitor the allocation in their memory pools or force them to release the allocated memory (for example at the end of a run):

  ```
  // Return the size of the total memory allocated for tracks
  //
  aTrackAllocator.GetAllocatedSize();

  // Return allocated storage for tracks to the free store
  //
  aTrackAllocator.ResetStorage();
  ```

- `G4ReferenceCountedHandle`
  Template class acting as a smart pointer and wrapping the type to be counted. It performs the reference counting during the life-time of the counted object.
- `G4FastVector`
  Template class defining a vector of pointers, not performing boundary checking.
- `G4PhysicsVector`
  Defines a physics vector which has values of energy-loss, cross-section, and other physics values of a particle in matter in a given range of the energy, momentum, etc. This class serves as the base class for a vector having various energy scale, for example like 'log' (`G4PhysicsLogVector`) 'linear' (`G4PhysicsLinearVector`), 'free' (`G4PhysicsFreeVector`), etc.
- `G4LPhysicsFreeVector`
  Implements a free vector for low energy physics cross-section data. A subdivision method is used to find the energy|momentum bin.
- `G4PhysicsOrderedFreeVector`
  A physics ordered free vector inherits from `G4PhysicsVector`. It provides, in addition, a method for the user to insert energy/value pairs in sequence. Methods to retrieve the max and min energies and values from the vector are also provided.
- `G4Timer`
  Utility class providing methods to measure elapsed user/system process time. Uses `<sys/times.h>` and `<unistd.h>` - POSIX.1.
- `G4UserLimits`
  Class collecting methods for get and set any kind of step limitation allowed in GEANT4.
- `G4UnitsTable`
  Placeholder for the system of units in GEANT4.

## 3.3 System of units

### 3.3.1 Basic units

GEANT4 offers the user the possibility to choose and use the preferred units for any quantity. In fact, GEANT4 takes care of the units. Internally a consistent set on units based on the `HepSystemOfUnits` is used:

```
millimeter                (mm)
nanosecond                (ns)
Mega electron Volt        (MeV)
positron charge           (eplus)
degree Kelvin             (kelvin)
the amount of substance   (mole)
luminous intensity        (candela)
radian                    (radian)
steradian                 (steradian)
```

All other units are defined from the basic ones.

For instance:

```
millimeter = mm = 1;
meter = m = 1000*mm;
...
m3 = m*m*m;
...
```

In the file `$CLHEP_BASE_DIR/include/CLHEP/Units/SystemOfUnits.h` from the CLHEP installation, one can find all units definitions.

One can also change the system of units to be used by the kernel.

### 3.3.2 Input your data

#### Avoid 'hard coded' data

The user **must** give the units for the data to introduce:

```
G4double Size = 15*km, KineticEnergy = 90.3*GeV, density = 11*mg/cm3;
```

GEANT4 assumes that these specifications for the units are respected, in order to assure independence from the units chosen in the client application.

If units are not specified in the client application, data are implicitly treated in internal GEANT4 system units; this practice is however strongly discouraged.

If the data set comes from an array or from an external file, it is strongly recommended to set the units as soon as the data are read, before any treatment. For instance:

```
for (int j=0, j<jmax, j++) CrossSection[j] *= millibarn;
...
my calculations
...
```

#### Interactive commands

Some built-in commands from the User Interface (UI) also require units to be specified.

For instance:

```
/gun/energy 15.2 keV
/gun/position 3 2 -7 meter
```

If units are not specified, or are not valid, the command is refused.

### 3.3.3 Output your data

You can output your data with the wished units. To do so, it is sufficient to **divide** the data by the corresponding unit:

```
G4cout << KineticEnergy/keV << " keV";
G4cout << density/(g/cm3)   << " g/cm3";
```

Of course, `G4cout << KineticEnergy` will print the energy in the internal units system.

There is another way to output the data. Let GEANT4 choose the most appropriate units for the actual numerical value of the data. It is sufficient to specify to which category the data belong to (Length, Time, Energy, etc.). For example:

```
G4cout << G4BestUnit(StepSize, "Length");
```

`StepSize` will be printed in km, m, mm, fermi, etc. depending of its actual value.

### 3.3.4 Introduce new units

If wished to introduce new units, there are two methods:

- You can extend the file `SystemOfUnits.h`

  ```
  #include "SystemOfUnits.h"

  static const G4double inch = 2.54*cm;
  ```

  Using this method, it is not easy to define composed units. It is better to do the following:
- Instantiate an object of the class `G4UnitDefinition`. These objects are owned by the global `G4UnitsTable` at construction, and must not be deleted by the user.

  ```
  new G4UnitDefinition ( name, symbol, category, value )
  ```

  For example: define a few units for speed

  ```
  new G4UnitDefinition ( "km/hour" , "km/h", "Speed", km/(3600*s) );
  new G4UnitDefinition ( "meter/ns", "m/ns", "Speed", m/ns );
  ```

  The category "Speed" does not exist by default in `G4UnitsTable`, but it will be created automatically. The class `G4UnitDefinition` is defined in `source/global/management/G4UnitsTable.hh`.

### 3.3.5 Print the list of units

You can print the list of units with the static function: `G4UnitDefinition::PrintUnitsTable();` or with the interactive command: `/units/list`

## 3.4 Run

### 3.4.1 Basic concept of *Run*

In GEANT4, *Run* is the largest unit of simulation. A run consists of a sequence of events. Within a run, the detector geometry, the set up of sensitive detectors, and the physics processes used in the simulation should be kept unchanged. A run is represented by a `G4Run` class object. A run starts with `BeamOn()` method of `G4RunManager`.

#### Representation of a run

`G4Run` represents a run. It has a run identification number, which should be set by the user, and the number of events simulated during the run. Please note that the run identification number is not used by the GEANT4 kernel, and thus can be arbitrarily assigned at the user's convenience.

`G4Run` has pointers to the tables `G4VHitsCollection` and `G4VDigiCollection`. These tables are associated in case *sensitive detectors* and *digitizer modules* are simulated, respectively. The usage of these tables will be mentioned in *Hits* and *Digitization*.

`G4Run` has two virtual methods, and thus you can extend `G4Run` class. In particular if you use GEANT4 in multi-threaded mode and need to accumulate values, these two virtual method must be overwritten to specify how such values should be collected firstly for a worker thread, and then for the entire run. These virtual methods are the following.

**`virtual void RecordEvent(const G4Event*)`** Method to be overwritten by the user for recording events in this (thread-local) run. At the end of the implementation, G4Run base-class method for must be invoked for recording data members in the base class.

**`void Merge(const G4Run*)`** Method to be overwritten by the user for merging local Run object to the global Run object. At the end of the implementation, G4Run base-class method for must be invoked for merging data members in the base class.

#### Manage the run procedures

`G4RunManager` manages the procedures of a run. In the constructor of `G4RunManager`, all of the manager classes in GEANT4 kernel, except for some static managers, are constructed. These managers are deleted in the destructor of `G4RunManager`. `G4RunManager` must be a singleton created in the user's `main()` program; the pointer to this singleton object can be obtained by other code using the `GetRunManager()` static method.

As already mentioned in *How to Define the main() Program*, all of the *user initialization* classes defined by the user should be assigned to `G4RunManager` before starting initialization of the GEANT4 kernel. The assignments of these user classes are done by `SetUserInitialization()` methods. All user classes defined by the GEANT4 kernel will be summarized in *User Actions*.

`G4RunManager` has several public methods, which are listed below.

**`Initialize()`** All initializations required by the GEANT4 kernel are triggered by this method. Initializations are:

- construction of the detector geometry and set up of sensitive detectors and/or digitizer modules,
- construction of particles and physics processes,
- calculation of cross-section tables.

This method is thus mandatory before proceeding to the first run. This method will be invoked automatically for the second and later runs in case some of the initialized quantities need to be updated.

**`BeamOn(G4int numberOfEvent)`** This method triggers the actual simulation of a run, that is, an event loop. It takes an integer argument which represents the number of events to be simulated.

**`GetRunManager()`** This static method returns the pointer to the G4RunManager singleton object.

**`GetCurrentEvent()`** This method returns the pointer to the G4Event object which is currently being simulated. This method is available only when an event is being processed. At this moment, the application state of GEANT4, which is explained in the following sub-section, is "EventProc". When GEANT4 is in a state other than "EventProc", this method returns null. Please note that the return value of this method is const G4Event * and thus you cannot modify the contents of the object.

**`SetNumberOfEventsToBeStored(G4int nPrevious)`** When simulating the "pile up" of more than one event, it is essential to access more than one event at the same moment. By invoking this method, G4RunManager keeps nPrevious G4Event objects. This method must be invoked before proceeding to BeamOn().

**`GetPreviousEvent(G4int i_thPrevious)`** The pointer to the i_thPrevious G4Event object can be obtained through this method. A pointer to a const object is returned. It is inevitable that i_thPrevious events must have already been simulated in the same run for getting the i_thPrevious event. Otherwise, this method returns null.

**`AbortRun()`** This method should be invoked whenever the processing of a run must be stopped. It is valid for GeomClosed and EventProc states. Run processing will be safely aborted even in the midst of processing an event. However, the last event of the aborted run will be incomplete and should not be used for further analysis.

#### Run manager classes for multi-threading mode

`G4MTRunManager` is the replacement of `G4RunManager` for multi-threading mode. At the very end of `Initialize()` method, `G4MTRunManager` creates and starts worker threads. The event each thread is tasked is in first-come-first-served basis, so that event numbers each thread has are not sequential.

`G4WorkerRunManager` is the local RunManager automatically instantiated by `G4MTRunManager` to take care of initialization and event handling of a thread. Both `G4MTRunManager` and `G4WorkerRunManager` are derived classes of `G4RunManager` base class.

The static method `G4RunManager::GetRunManager()` returns the following pointer.

- It returns the pointer to the `G4WorkerRunManager` of the local thread when it is invoked from thread-local object.
- It returns the pointer to the `G4MTRunManager` when it is invoked from shared object.
- It returns the pointer to the base `G4RunManager` if it is used in the sequential mode.

`G4RunManager` has a method `GetRunManagerType()` that returns an enum named `RMType` to indicate what kind of `RunManager` it is. `RMType` is defined as `{ sequentialRM, masterRM, workerRM }`. From the thread-local object, a static method `G4MTRunManager::GetMasterRunManager()` is available to access to `G4MTRunManager`. From a worker thread, the user may access to, for example, detector construction (it is a shared class) through this `GetMasterRunManager()` method.

#### G4UserRunAction

G4UserRunAction is one of the *user action* classes from which you can derive your own concrete class. This base class has three virtual methods as follows:

**GenerateRun()** This method is invoked at the beginning of the BeamOn() method but after confirmation of the conditions of the GEANT4 kernel. This method should be used to instantiate a user-specific run class object.

**BeginOfRunAction()** This method is invoked at the beginning of the BeamOn() method but after confirmation of the conditions of the GEANT4 kernel. Likely uses of this method include:

- setting a run identification number,
- booking histograms,
- setting run specific conditions of the sensitive detectors and/or digitizer modules (e.g., dead channels).

**EndOfRunAction()** This method is invoked at the very end of the BeamOn() method. Typical use cases of this method are

- store/print histograms,
- manipulate run summaries.

### 3.4.2 GEANT4 as a state machine

GEANT4 is designed as a state machine. Some methods in GEANT4 are available for only a certain state(s). G4RunManager controls the state changes of the GEANT4 application. States of GEANT4 are represented by the enumeration G4ApplicationState. It has six states through the life cycle of a GEANT4 application.

**G4State_PreInit state** A GEANT4 application starts with this state. The application needs to be initialized when it is in this state. The application occasionally comes back to this state if geometry, physics processes, and/or cut-off have been changed after processing a run.

**G4State_Init state** The application is in this state while the Initialize() method of G4RunManager is being invoked. Methods defined in any user initialization classes are invoked during this state.

**G4State_Idle state** The application is ready for starting a run.

**G4State_GeomClosed state** When BeamOn() is invoked, the application proceeds to this state to process a run. Geometry, physics processes, and cut-off cannot be changed during run processing.

**G4State_EventProc state** A GEANT4 application is in this state when a particular event is being processed. GetCurrentEvent() and GetPreviousEvent() methods of G4RunManager are available only at this state.

**G4State_Quit state** When the destructor of G4RunManager is invoked, the application comes to this "dead end" state. Managers of the GEANT4 kernel are being deleted and thus the application cannot come back to any other state.

**G4State_Abort state** When a G4Exception occurs, the application comes to this "dead end" state and causes a core dump. The user still has a hook to do some "safe" operations, e.g. storing histograms, by implementing a user concrete class of G4VStateDependent. The user also has a choice to suppress the occurrence of G4Exception by a UI command /control/suppressAbortion. When abortion is suppressed, you will still get error messages issued by G4Exception, and there is NO guarantee of a correct result after the G4Exception error message.

G4StateManager belongs to the *intercoms* category.

### 3.4.3 User's hook for state change

In case the user wants to do something at the moment of state change of GEANT4, the user can create a concrete class of the `G4VStateDependent` base class. For example, the user can store histograms when G4Exception occurs and GEANT4 comes to the *Abort* state, but before the actual core dump.

The following is an example user code which stores histograms when GEANT4 becomes to the *Abort* state. This class object should be made in, for example `main()`, by the user code. This object will be automatically registered to `G4StateManager` at its construction.

Listing 3.1: Header file of UserHookForAbortState

```
#ifndef UserHookForAbortState_H
#define UserHookForAbortState_H 1

#include "G4VStateDependent.hh"

class UserHookForAbortState : public G4VStateDependent
{
 public:
  UserHookForAbortState();   // constructor
  ~UserHookForAbortState();  // destructor

  virtual G4bool Notify(G4ApplicationState requiredState);
};
```

Listing 3.2: Source file of UserHookForAbortState

```
#include "UserHookForAbortState.hh"

UserHookForAbortState::UserHookForAbortState() {;}
UserHookForAbortState::~UserHookForAbortState() {;}

G4bool UserHookForAbortState::Notify(G4ApplicationState requiredState)
{
  if(requiredState!=Abort) return true;

  // Do book keeping here

  return true;
}
```

### 3.4.4 Customizing the Run Manager

#### Virtual Methods in the Run Manager

`G4RunManager` is a concrete class with a complete set of functionalities for managing the GEANT4 kernel. It is the only manager class in the GEANT4 kernel which must be constructed in the `main()` method of the user's application. Thus, instead of constructing the `G4RunManager` provided by GEANT4, you are free to construct your own `RunManager`. It is recommended, however, that your `RunManager` inherit `G4RunManager`. For this purpose, `G4RunManager` has various virtual methods which provide all the functionalities required to handle the GEANT4 kernel. Hence, your customized run manager need only override the methods particular to your needs; the remaining methods in `G4RunManager` base class can still be used. A summary of the available methods is presented here:

**`public: virtual void Initialize();`** main entry point of GEANT4 kernel initialization
**`protected: virtual void InitializeGeometry();`** geometry construction
**`protected: virtual void InitializePhysics();`** physics processes construction
**`public: virtual void BeamOn(G4int n_event);`** main entry point of the event loop

**protected: virtual G4bool ConfirmBeamOnCondition();** check the kernel conditions for the event loop

**protected: virtual void RunInitialization();** prepare a run

**protected: virtual void DoEventLoop(G4int n_events);** manage an event loop

**protected: virtual G4Event* GenerateEvent(G4int i_event);** generation of G4Event object

**protected: virtual void AnalyzeEvent(G4Event* anEvent);** storage/analysis of an event

**protected: virtual void RunTermination();** terminate a run

**public: virtual void DefineWorldVolume(G4VPhysicalVolume * worldVol);** set the world volume to G4Navigator

**public: virtual void AbortRun();** abort the run

### Customizing the Event Loop

In `G4RunManager` the event loop is handled by the virtual method `DoEventLoop()`. This method is implemented by a `for` loop consisting of the following steps:

1. construct a `G4Event` object and assign to it primary vertex(es) and primary particles. This is done by the virtual `GeneratePrimaryEvent()` method.
2. send the `G4Event` object to `G4EventManager` for the detector simulation. *Hits* and *trajectories* will be associated with the `G4Event` object as a consequence.
3. perform bookkeeping for the current `G4Event` object. This is done by the virtual `AnalyzeEvent()` method.

`DoEventLoop()` performs the entire simulation of an event. However, it is often useful to split the above three steps into isolated application programs. If, for example, you wish to examine the effects of changing discriminator thresholds, ADC gate widths and/or trigger conditions on simulated events, much time can be saved by performing steps 1 and 2 in one program and step 3 in another. The first program need only generate the hit/trajectory information once and store it, perhaps in a database. The second program could then retrieve the stored `G4Event` objects and perform the digitization (analysis) using the above threshold, gate and trigger settings. These settings could then be changed and the digitization program re-run without re-generating the `G4Events`.

### Changing the Detector Geometry

The detector geometry defined in your `G4VUserDetectorConstruction` concrete class can be changed during a run break (between two runs). Two different cases are considered.

The first is the case in which you want to delete the entire structure of your old geometry and build up a completely new set of volumes. For this case, you need to delete them by yourself, and let *RunManager* invokes `Construct()` and `ConstructSDandField()` methods of your detector construction once again when `RunManager` starts the next run.

```
G4RunManager* runManager = G4RunManager::GetRunManager();
runManager->ReinitializeGeometry();
```

If this `ReinitializeGeometry()` is invoked, `GeometryHasBeenModified()` (discussed next) is automatically invoked. Presumably this case is rather rare. The second case is more frequent for the user.

The second case is the following. Suppose you want to move and/or rotate a particular piece of your detector component. This case can easily happen for a beam test of your detector. It is obvious for this case that you need not change the world volume. Rather, it should be said that your world volume (experimental hall for your beam test) should be big enough for moving/rotating your test detector. For this case, you can still use all of your detector geometries, and just use a `Set` method of a particular physical volume to update the transformation vector as you want. Thus, you don't need to re-set your world volume pointer to *RunManager*.

If you want to change your geometry for every run, you can implement it in the `BeginOfRunAction()` method of `G4UserRunAction` class, which will be invoked at the beginning of each run, or, derive the

`RunInitialization()` method. Please note that, for both of the above mentioned cases, you need to let *RunManager* know "the geometry needs to be closed again". Thus, you need to invoke

```
runManager->GeometryHasBeenModified();
```

before proceeding to the next run. An example of changing geometry is given in a GEANT4 tutorial in GEANT4 Training kit #2.

### Switch physics processes

In the `InitializePhysics()` method, `G4VUserPhysicsList::Construct` is invoked in order to define particles and physics processes in your application. Basically, you can not add nor remove any particles during execution, because particles are static objects in GEANT4 (see *How to Specify Particles* and *Particles* for details). In addition, it is very difficult to add and/or remove physics processes during execution, because registration procedures are very complex, except for experts (see *How to Specify Physics Processes* and *Physics Processes*). This is why the `initializePhysics()` method is assumed to be invoked at once in GEANT4 kernel initialization.

However, you can switch on/off physics processes defined in your `G4VUserPhysicsList` concrete class and also change parameters in physics processes during the run break.

You can use `ActivateProcess()` and `InActivateProcess()` methods of `G4ProcessManager` anywhere outside the event loop to switch on/off some process. You should be very careful to switch on/off processes inside the event loop, though it is not prohibited to use these methods even in the *EventProc* state.

It is a likely case to change cut-off values in a run. You can change `defaultCutValue` in `G4VUserPhysicsList` during the *Idle* state. In this case, all cross section tables need to be recalculated before the event loop. You should use the `CutOffHasBeenModified()` method when you change cut-off values so that the `SetCuts` method of your *PhysicsList* concrete class will be invoked.

## 3.4.5 Managing worker thread

`G4UserWorkerInitialization` is an additional user initialization class to be used only for the multi-threaded mode. The object of this class can be set to `G4MTRunManager`, but not to `G4RunManager`. `G4UserWorkerInitialization` class has five virtual methods as the user hooks which are invoked at several occasions of the life cycle of each thread.

**`virtual void WorkerInitialize() const`** This method is called after the tread is created but before the G4WorkerRunManager is instantiated.

**`virtual void WorkerStart() const`** This method is called once at the beginning of simulation job when kernel classes and user action classes have already instantiated but geometry and physics have not been yet initialized. This situation is identical to "PreInit" state in the sequential mode.

**`virtual void WorkerStartRun() const`** This method is called before an event loop. Geometry and physics have already been set up for the thread. All threads are synchronized and ready to start the local event loop. This situation is identical to "Idle" state in the sequential mode.

**`virtual void WorkerRunEnd() const`** This method is called for each thread when the local event loop is done, but before the synchronization over all worker threads.

**`virtual void WorkerStop() const`** This method is called once at the end of simulation job.

## 3.5 Event

### 3.5.1 Representation of an event

`G4Event` represents an event. An object of this class contains all inputs and outputs of the simulated event. This class object is constructed in `G4RunManager` and sent to `G4EventManager`. The event currently being processed can be obtained via the `getCurrentEvent()` method of `G4RunManager`.

### 3.5.2 Structure of an event

A `G4Event` object has four major types of information. Get methods for this information are available in `G4Event`.

**Primary vertexes and primary particles** Details are given in *Event Generator Interface*.

**Trajectories** Trajectories are stored in G4TrajectoryContainer class objects and the pointer to this container is stored in `G4Event`. The contents of a trajectory are given in *Trajectory and Trajectory Point*.

**Hits collections** Collections of hits generated by *sensitive detectors* are kept in `G4HCofThisEvent` class object and the pointer to this container class object is stored in `G4Event`. See *Hits* for the details.

**Digits collections** Collections of digits generated by *digitizer modules* are kept in `G4DCofThisEvent` class object and the pointer to this container class object is stored in `G4Event`. See *Digitization* for the details.

### 3.5.3 Mandates of `G4EventManager`

`G4EventManager` is the manager class to take care of one event. It is responsible for:

- converting `G4PrimaryVertex` and `G4PrimaryParticle` objects associated with the current `G4Event` object to `G4Track` objects. All of `G4Track` objects representing the primary particles are sent to `G4StackManager`.
- Pop one `G4Track` object from `G4StackManager` and send it to `G4TrackingManager`. The current `G4Track` object is deleted by `G4EventManager` after the track is simulated by `G4TrackingManager`, if the track is marked as “killed”.
- In case the primary track is “suspended” or “postponed to next event” by `G4TrackingManager`, it is sent back to the `G4StackManager`. Secondary `G4Track` objects returned by `G4TrackingManager` are also sent to `G4StackManager`.
- When `G4StackManager` returns `NULL` for the “pop” request, `G4EventManager` terminates the current processing event.
- invokes the user-defined methods `beginOfEventAction()` and `endOfEventAction()` from the `G4UserEventAction` class. See *User Information Classes* for details.

### 3.5.4 Stacking mechanism

`G4StackManager` has three stacks, named *urgent*, *waiting* and *postpone-to-next-event*, which are objects of the `G4TrackStack` class. By default, all `G4Track` objects are stored in the *urgent* stack and handled in a “last in first out” manner. In this case, the other two stacks are not used. However, tracks may be routed to the other two stacks by the user-defined `G4UserStackingAction` concrete class.

If the methods of `G4UserStackingAction` have been overridden by the user, the *postpone-to-next-event* and *waiting* stacks may contain tracks. At the beginning of an event, `G4StackManager` checks to see if any tracks left over from the previous event are stored in the *postpone-to-next-event stack*. If so, it attempts to move them to the *urgent* stack. But first the `PrepareNewEvent()` method of `G4UserStackingAction` is called. Here tracks may be re-classified by the user and sent to the *urgent* or *waiting* stacks, or deferred again to the *postpone-to-next-event* stack. As the event is processed `G4StackManager` pops tracks from the *urgent* stack until it is empty. At this point the `NewStage()` method of `G4UserStackingAction` is called. In this method tracks from the *waiting* stack may be sent to the *urgent* stack, retained in the *waiting* stack or postponed to the next event.

Details of the user-defined methods of `G4UserStackingAction` and how they affect track stack management are given in *User Information Classes*.

## 3.6 Event Generator Interface

### 3.6.1 Structure of a primary event

#### Primary vertex and primary particle

The `G4Event` class object should have a set of primary particles when it is sent to `G4EventManager` via `processOneEvent()` method. It is the mandate of your `G4VUserPrimaryGeneratorAction` concrete class to send primary particles to the `G4Event` object.

The `G4PrimaryParticle` class represents a primary particle with which GEANT4 starts simulating an event. This class object has information on particle type and its three momenta. The positional and time information of primary particle(s) are stored in the `G4PrimaryVertex` class object and, thus, this class object can have one or more `G4PrimaryParticle` class objects which share the same vertex. Primary vertexes and primary particles are associated with the `G4Event` object by a form of linked list.

A concrete class of `G4VPrimaryGenerator`, the `G4PrimaryParticle` object is constructed with either a pointer to `G4ParticleDefinition` or an integer number which represents P.D.G. particle code. For the case of some artificial particles, e.g., geantino, optical photon, etc., or exotic nuclear fragments, which the P.D.G. particle code does not cover, the `G4PrimaryParticle` should be constructed by `G4ParticleDefinition` pointer. On the other hand, elementary particles with very short life time, e.g., weak bosons, or quarks/gluons, can be instantiated as `G4PrimaryParticle` objects using the P.D.G. particle code. It should be noted that, even though primary particles with such a very short life time are defined, GEANT4 will simulate only the particles which are defined as `G4ParticleDefinition` class objects. Other primary particles will be simply ignored by `G4EventManager`. But it may still be useful to construct such "intermediate" particles for recording the origin of the primary event.

#### Forced decay channel

The `G4PrimaryParticle` class object can have a list of its daughter particles. If the parent particle is an "intermediate" particle, which GEANT4 does not have a corresponding `G4ParticleDefinition`, this parent particle is ignored and daughters are assumed to start from the vertex with which their parent is associated. For example, a Z boson is associated with a vertex and it has positive and negative muons as its daughters, these muons will start from that vertex.

There are some kinds of particles which should fly some reasonable distances and, thus, should be simulated by GEANT4, but you still want to follow the decay channel generated by an event generator. A typical case of these particles is B meson. Even for the case of a primary particle which has a corresponding `G4ParticleDefinition`, it can have daughter primary particles. GEANT4 will trace the parent particle until it comes to decay, obeying multiple scattering, ionization loss, rotation with the magnetic field, etc. according to its particle type. When the parent comes to decay, instead of randomly choosing its decay channel, it follows the "pre-assigned" decay channel. To conserve the energy and the momentum of the parent, daughters will be Lorentz transformed according to their parent's frame.

### 3.6.2 Interface to a primary generator

#### G4HEPEvtInterface

Unfortunately, almost all event generators presently in use, commonly are written in FORTRAN. For GEANT4, it was decided to not link with any FORTRAN program or library, even though the C++ language syntax itself allows such a link. Linking to a FORTRAN package might be convenient in some cases, but we will lose many advantages of object-oriented features of C++, such as robustness. Instead, GEANT4 provides an ASCII file interface for such event generators.

`G4HEPEvtInterface` is one of `G4VPrimaryGenerator` concrete class and thus it can be used in your `G4VUserPrimaryGeneratorAction` concrete class. `G4HEPEvtInterface` reads an ASCII file produced by an event generator and reproduces `G4PrimaryParticle` objects associated with a `G4PrimaryVertex` object. It reproduces a full production chain of the event generator, starting with primary quarks, etc. In other words, `G4HEPEvtInterface` converts information stored in the `/HEPEVT/` common block to an object-oriented data structure. Because the `/HEPEVT/` common block is commonly used by almost all event generators written in FORTRAN, `G4HEPEvtInterface` can interface to almost all event generators currently used in the HEP community. The constructor of `G4HEPEvtInterface` takes the file name. Listing 3.3 shows an example how to use `G4HEPEvtInterface`. Note that an event generator is not assumed to give a place of the primary particles, the interaction point must be set before invoking `GeneratePrimaryVertex()` method.

Listing 3.3: An example code for `G4HEPEvtInterface`

```
#ifndef ExN04PrimaryGeneratorAction_h
#define ExN04PrimaryGeneratorAction_h 1

#include "G4VUserPrimaryGeneratorAction.hh"
#include "globals.hh"

class G4VPrimaryGenerator;
class G4Event;

class ExN04PrimaryGeneratorAction : public G4VUserPrimaryGeneratorAction
{
  public:
    ExN04PrimaryGeneratorAction();
    ~ExN04PrimaryGeneratorAction();

  public:
    void GeneratePrimaries(G4Event* anEvent);

  private:
    G4VPrimaryGenerator* HEPEvt;
};

#endif


#include "ExN04PrimaryGeneratorAction.hh"

#include "G4Event.hh"
#include "G4HEPEvtInterface.hh"

ExN04PrimaryGeneratorAction::ExN04PrimaryGeneratorAction()
{
  HEPEvt = new G4HEPEvtInterface("pythia_event.data");
}

ExN04PrimaryGeneratorAction::~ExN04PrimaryGeneratorAction()
{
  delete HEPEvt;
}
```

(continues on next page)

(continued from previous page)

```
void ExN04PrimaryGeneratorAction::GeneratePrimaries(G4Event* anEvent)
{
  HEPEvt->SetParticlePosition(G4ThreeVector(0.*cm,0.*cm,0.*cm));
  HEPEvt->GeneratePrimaryVertex(anEvent);
}
```

### Format of the ASCII file

An ASCII file, which will be fed by `G4HEPEvtInterface` should have the following format.

- The first line of each primary event should be an integer which represents the number of the following lines of primary particles.
- Each line in an event corresponds to a particle in the `/HEPEVT/` common. Each line has `ISTHEP, IDHEP, JDAHEP(1), JDAHEP(2), PHEP(1), PHEP(2), PHEP(3), PHEP(5)`. Refer to the `/HEPEVT/` manual for the meanings of these variables.

Listing 3.4 shows an example FORTRAN code to generate an ASCII file.

Listing 3.4: A FORTRAN example using the `/HEPEVT/` common.

```
***********************************************************
      SUBROUTINE HEP2G4
*
* Convert /HEPEVT/ event structure to an ASCII file
* to be fed by G4HEPEvtInterface
*
***********************************************************
      PARAMETER (NMXHEP=2000)
      COMMON/HEPEVT/NEVHEP,NHEP,ISTHEP(NMXHEP),IDHEP(NMXHEP),
     >JMOHEP(2,NMXHEP),JDAHEP(2,NMXHEP),PHEP(5,NMXHEP),VHEP(4,NMXHEP)
      DOUBLE PRECISION PHEP,VHEP
*
      WRITE(6,*) NHEP
      DO IHEP=1,NHEP
       WRITE(6,10)
     >  ISTHEP(IHEP),IDHEP(IHEP),JDAHEP(1,IHEP),JDAHEP(2,IHEP),
     >  PHEP(1,IHEP),PHEP(2,IHEP),PHEP(3,IHEP),PHEP(5,IHEP)
10     FORMAT(4I10,4(1X,D15.8))
      ENDDO
*
      RETURN
      END
```

### Future interface to the new generation generators

Several activities have already been started for developing object-oriented event generators. Such new generators can be easily linked and used with a GEANT4 based simulation. Furthermore, we need not distinguish a primary generator from the physics processes used in GEANT4. Future generators can be a kind of physics process plugged-in by inheriting `G4VProcess`.

### 3.6.3 Event overlap using multiple generators

Your `G4VUserPrimaryGeneratorAction` concrete class can have more than one `G4VPrimaryGenerator` concrete class. Each `G4VPrimaryGenerator` concrete class can be accessed more than once per event. Using these class objects, one event can have more than one primary event.

One possible use is the following. Within an event, a `G4HEPEvtInterface` class object instantiated with a minimum bias event file is accessed 20 times and another `G4HEPEvtInterface` class object instantiated with a signal event file is accessed once. Thus, this event represents a typical signal event of LHC overlapping 20 minimum bias events. It should be noted that a simulation of event overlapping can be done by merging hits and/or digits associated with several events, and these events can be simulated independently. Digitization over multiple events will be mentioned in *Digitization*.

## 3.7 Event Biasing Techniques

### 3.7.1 Scoring, Geometrical Importance Sampling and Weight Roulette

GEANT4 provides event biasing techniques which may be used to save computing time in such applications as the simulation of radiation shielding. These are *geometrical splitting* and *Russian roulette* (also called geometrical importance sampling), and *weight roulette*. Scoring is carried out by `G4MultiFunctionalDetector` (see *G4MultiFunctionalDetector and G4VPrimitiveScorer* and *Concrete classes of G4VPrimitiveScorer*) using the standard GEANT4 scoring technique. Biasing specific scorers have been implemented and are described within `G4MultiFunctionalDetector` documentation. In this chapter, it is assumed that the reader is familiar with both the usage of GEANT4 and the concepts of importance sampling. More detailed documentation may be found in the documents 'Scoring, geometrical importance sampling and weight roulette'.

A detailed description of different use-cases which employ the sampling and scoring techniques can be found in the document 'Use cases of importance sampling and scoring in Geant4'.

The purpose of importance sampling is to save computing time by sampling less often the particle histories entering "less important" geometry regions, and more often in more "important" regions. Given the same amount of computing time, an importance-sampled and an analogue-sampled simulation must show equal mean values, while the importance-sampled simulation will have a decreased variance.

The implementation of scoring is independent of the implementation of importance sampling. However both share common concepts. *Scoring and importance sampling apply to particle types chosen by the user*, which should be borne in mind when interpreting the output of any biased simulation.

Examples on how to use scoring and importance sampling may be found in `examples/extended/biasing`.

#### Geometries

The kind of scoring referred to in this note and the importance sampling apply to spatial cells provided by the user.

A **cell** is a physical volume (further specified by it's replica number, if the volume is a replica). Cells may be defined in two kinds of geometries:

1. **mass geometry**: the geometry setup of the experiment to be simulated. Physics processes apply to this geometry.
2. **parallel-geometry**: a geometry constructed to define the physical volumes according to which scoring and/or importance sampling is applied.

The user has the choice to score and/or sample by importance the particles of the chosen type, according to mass geometry or to parallel geometry. It is possible to utilize several parallel geometries in addition to the mass geometry. This provides the user with a lot of flexibility to define separate geometries for different particle types in order to apply scoring or/and importance sampling.

**Note:** Parallel geometries should be constructed using the implementation as described in *Parallel Geometries*. There are a few conditions for parallel geometries:

- The world volume for parallel and mass geometries must be identical copies.
- Scoring and importance cells must not share boundaries with the world volume.

### Changing the Sampling

Samplers are higher level tools which perform the necessary changes of the GEANT4 sampling in order to apply importance sampling and weight roulette.

Variance reduction (and scoring through the `G4MultiFunctionalDetector`) may be combined arbitrarily for chosen particle types and may be applied to the mass or to parallel geometries.

The `G4GeometrySampler` can be applied equally to mass or parallel geometries with an abstract interface supplied by `G4VSampler`. `G4VSampler` provides `Prepare...` methods and a `Configure` method:

```
class G4VSampler
{
  public:
   G4VSampler();
   virtual ~G4VSampler();
   virtual void PrepareImportanceSampling(G4VIStore *istore,
                                           const G4VImportanceAlgorithm
                                           *ialg = 0) = 0;
   virtual void PrepareWeightRoulett(G4double wsurvive = 0.5,
                                      G4double wlimit = 0.25,
                                      G4double isource = 1) = 0;
   virtual void PrepareWeightWindow(G4VWeightWindowStore *wwstore,
                                     G4VWeightWindowAlgorithm *wwAlg = 0,
                                     G4PlaceOfAction placeOfAction =
                                     onBoundary) = 0;
   virtual void Configure() = 0;
   virtual void ClearSampling() = 0;
   virtual G4bool IsConfigured() const = 0;
};
```

The methods for setting up the desired combination need specific information:

- Importance sampling: message `PrepareImportanceSampling` with a `G4VIStore` and optionally a `G4VImportanceAlgorithm`
- Weight window: message `PrepareWeightWindow` with the arguments:
  - **wwstore*: a `G4VWeightWindowStore` for retrieving the lower weight bounds for the energy-space cells
  - **wwAlg*: a `G4VWeightWindowAlgorithm` if a customized algorithm should be used
  - *placeOfAction*: a `G4PlaceOfAction` specifying where to perform the biasing
- Weight roulette: message `PrepareWeightRoulett` with the optional parameters:
  - *wsurvive*: survival weight
  - *wlimit*: minimal allowed value of weight * source importance / cell importance
  - *isource*: importance of the source cell

Each object of a sampler class is responsible for one particle type. The particle type is given to the constructor of the sampler classes via the particle type name, e.g. “neutron”. Depending on the specific purpose, the `Configure()` of a sampler will set up specialized processes (derived from `G4VProcess`) for transportation in the parallel geometry, importance sampling and weight roulette for the given particle type. When `Configure()` is invoked the sampler places the processes in the correct order independent of the order in which user invoked the `Prepare...` methods.

**Note:**

- The `Prepare...()` functions may each only be invoked once.
- To configure the sampling the function `Configure()` must be called *after* the `G4RunManager` has been initialized and the PhysicsList has been instantiated.

The interface and framework are demonstrated in the `examples/extended/biasing` directory, with the main changes being to the G4GeometrySampler class and the fact that in the parallel case the WorldVolume is a copy of the Mass World. The parallel geometry now has to inherit from `G4VUserParallelWorld` which also has the `GetWorld()` method in order to retrieve a copy of the mass geometry WorldVolume.

```
class B02ImportanceDetectorConstruction : public G4VUserParallelWorld
ghostWorld = GetWorld();
```

The constructor for `G4GeometrySampler` takes a pointer to the physical world volume and the particle type name (e.g. "neutron") as arguments. In a single mass geometry the sampler is created as follows:

```
G4GeometrySampler mgs(detector->GetWorldVolume(),"neutron");
mgs.SetParallel(false);
```

Whilst the following lines of code are required in order to set up the sampler for the parallel geometry case:

```
G4VPhysicalVolume* ghostWorld = pdet->GetWorldVolume();

G4GeometrySampler pgs(ghostWorld,"neutron");

pgs.SetParallel(true);
```

Also note that the preparation and configuration of the samplers has to be carried out *after* the instantiation of the UserPhysicsList. With the modular reference PhysicsList the following set-up is required (first is for biasing, the second for scoring):

```
physicsList->RegisterPhysics(new G4ImportanceBiasing(&pgs,parallelName));
physicsList->RegisterPhysics(new G4ParallelWorldPhysics(parallelName));
```

If the a UserPhysicsList is being implemented, then the following should be used to give the pointer to the GeometrySampler to the PhysicsList:

```
physlist->AddBiasing(&pgs,parallelName);
```

Then to instantiate the biasing physics process the following should be included in the UserPhysicsList and called from `ConstructProcess()`:

```
AddBiasingProcess(){
  fGeomSampler->SetParallel(true); // parallelworld
  G4IStore* iStore = G4IStore::GetInstance(fBiasWorldName);
  fGeomSampler->SetWorld(iStore->GetParallelWorldVolumePointer());
  //  fGeomSampler->PrepareImportanceSampling(G4IStore::
  //                               GetInstance(fBiasWorldName), 0);
  static G4bool first = true;
  if(first) {
    fGeomSampler->PrepareImportanceSampling(iStore, 0);

    fGeomSampler->Configure();
    G4cout << " GeomSampler Configured!!! " << G4endl;
    first = false;
  }

#ifdef G4MULTITHREADED
```

(continues on next page)

(continued from previous page)

```
    fGeomSampler->AddProcess();
#else
  G4cout << " Running in singlethreaded mode!!! " << G4endl;
#endif
```

```
pgs.PrepareImportanceSampling(G4IStore::GetInstance(pdet->GetName()), 0);
pgs.Configure();
```

Due to the fact that biasing is a process and has to be inserted after all the other processes have been created.

## Importance Sampling

Importance sampling acts on particles crossing boundaries between "importance cells". The action taken depends on the importance values assigned to the cells. In general a particle history is either split or Russian roulette is played if the importance increases or decreases, respectively. A weight assigned to the history is changed according to the action taken.

The tools provided for importance sampling require the user to have a good understanding of the physics in the problem. This is because the user has to decide which particle types require importance sampled, define the cells, and assign importance values to the cells. If this is not done properly the results cannot be expected to describe a real experiment.

The assignment of importance values to a cell is done using an importance store described below.

An "importance store" with the interface `G4VIStore` is used to store importance values related to cells. In order to do importance sampling the user has to create an object (e.g. of class `G4IStore`) of type `G4VIStore`. The samplers may be given a `G4VIStore`. The user fills the store with cells and their importance values. The store is now a singleton class so should be created using a GetInstance method:

```
G4IStore *aIstore = G4IStore::GetInstance();
```

Or if a parallel world is used:

```
G4IStore *aIstore = G4IStore::GetInstance(pdet->GetName());
```

An importance store has to be constructed with a reference to the world volume of the geometry used for importance sampling. This may be the world volume of the mass or of a parallel geometry. Importance stores derive from the interface `G4VIStore`:

```
class  G4VIStore
{
  public:
    G4VIStore();
    virtual  ~G4VIStore();
    virtual G4double GetImportance(const G4GeometryCell &gCell) const = 0;
    virtual G4bool IsKnown(const G4GeometryCell &gCell) const = 0;
    virtual const G4VPhysicalVolume &GetWorldVolume() const = 0;
};
```

A concrete implementation of an importance store is provided by the class `G4VStore`. The *public* part of the class is:

```
class G4IStore : public G4VIStore
{
  public:
    explicit G4IStore(const G4VPhysicalVolume &worldvolume);
    virtual ~G4IStore();
    virtual G4double GetImportance(const G4GeometryCell &gCell) const;
    virtual G4bool IsKnown(const G4GeometryCell &gCell) const;
```

(continues on next page)

(continued from previous page)

```
    virtual const G4VPhysicalVolume &GetWorldVolume() const;
    void AddImportanceGeometryCell(G4double importance,
                                    const G4GeometryCell &gCell);
    void AddImportanceGeometryCell(G4double importance,
                                    const G4VPhysicalVolume &,
                                          G4int aRepNum = 0);
    void ChangeImportance(G4double importance,
                                 const G4GeometryCell &gCell);
    void ChangeImportance(G4double importance,
                                 const G4VPhysicalVolume &,
                                       G4int aRepNum = 0);
    G4double GetImportance(const G4VPhysicalVolume &,
                                        G4int aRepNum = 0) const ;
  private: .....
};
```

The member function `AddImportanceGeometryCell()` enters a cell and an importance value into the importance store. The importance values may be returned either according to a physical volume and a replica number or according to a `G4GeometryCell`. The user must be aware of the interpretation of assigning importance values to a cell. If scoring is also implemented then this is attached to logical volumes, in which case the physical volume and replica number method should be used for assigning importance values. See `examples/extended/biasing` `B01` and `B02` for examples of this.

**Note:** An importance value must be assigned to every cell.

The different cases:

- *Cell is not in store*
  Not filling a certain cell in the store will cause an exception.
- *Importance value = zero*
  Tracks of the chosen particle type will be killed.
- *importance values > 0*
  Normal allowed values
- *Importance value smaller zero*
  Not allowed!

### The Importance Sampling Algorithm

Importance sampling supports using a customized importance sampling algorithm. To this end, the sampler interface *Changing the Sampling* may be given a pointer to the interface `G4VImportanceAlgorithm`:

```
class G4VImportanceAlgorithm
{
  public:
    G4VImportanceAlgorithm();
    virtual ~G4VImportanceAlgorithm();
    virtual G4Nsplit_Weight Calculate(G4double ipre,
                                       G4double ipost,
                                       G4double init_w) const = 0;
};
```

The method `Calculate()` takes the arguments:

- *ipre*, *ipost* : importance of the previous cell and the importance of the current cell, respectively.
- *init_w*: the particle's weight

It returns the struct:

```
class G4Nsplit_Weight
{
  public:

  G4int fN;
  G4double fW;
};
```

- *fN*: the calculated number of particles to exit the importance sampling
- *fW*: the weight of the particles

The user may have a customized algorithm used by providing a class inheriting from `G4VImportanceAlgorithm`.

If no customized algorithm is given to the sampler the default importance sampling algorithm is used. This algorithm is implemented in `G4ImportanceAlgorithm`.

### The Weight Window Technique

The weight window technique is a weight-based alternative to importance sampling:

- applies splitting and Russian roulette depending on space (cells) and energy
- user defines weight windows in contrast to defining importance values as in importance sampling

In contrast to importance sampling this technique is not weight blind. Instead the technique is applied according to the particle weight with respect to the current energy-space cell.

Therefore the technique is convenient to apply in combination with other variance reduction techniques such as cross-section biasing and implicit capture.

A weight window may be specified for every cell and for several energy regions: *space-energy cell*.

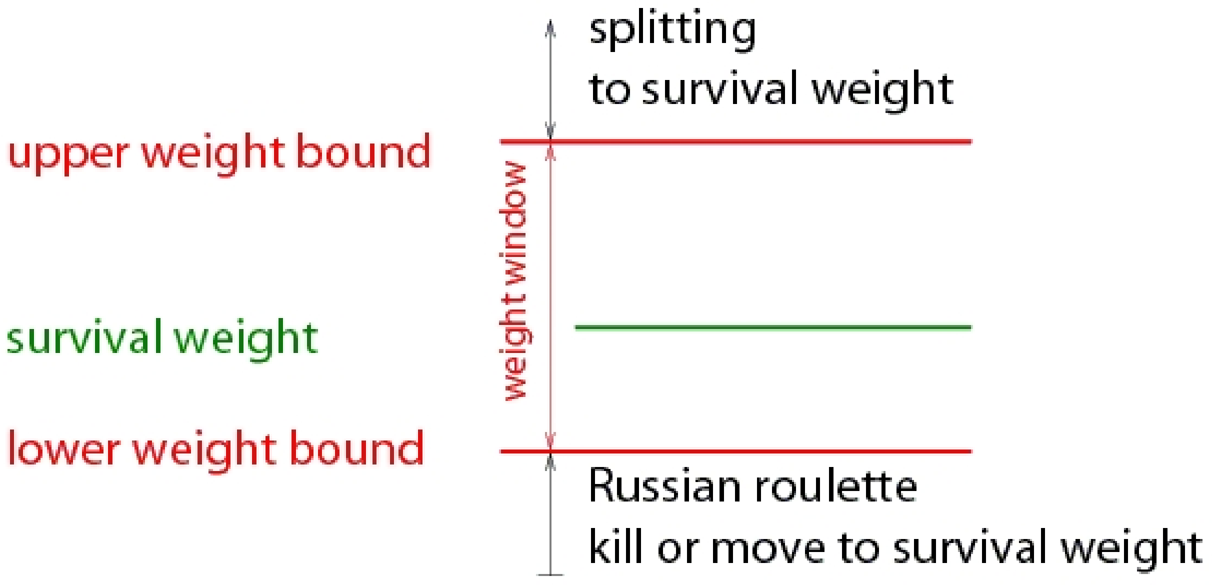


Fig. 3.2: Weight window concept

**Weight window concept**

The user specifies a *lower weight bound W_L* for every space-energy cell.

- The upper weight bound W_U and the survival weight W_S are calculated as:

W_U = C_U *W_L* and
W_S = C_S *W_L*.

- The user specifies C_S and C_U once for the whole problem.
- The user may give different sets of energy bounds for every cell or one set for all geometrical cells
- Special case: if C_S = C_U = 1 for all energies then weight window is equivalent to importance sampling
- The user can choose to apply the technique: at boundaries, on collisions or on boundaries and collisions

The energy-space cells are realized by `G4GeometryCell` as in importance sampling. The cells are stored in a weight window store defined by `G4VWeightWindowStore`:

```
class  G4VWeightWindowStore {
 public:
   G4VWeightWindowStore();
   virtual  ~G4VWeightWindowStore();
   virtual G4double GetLowerWeitgh(const G4GeometryCell &gCell,
                                  G4double partEnergy) const = 0;
   virtual G4bool IsKnown(const G4GeometryCell &gCell) const = 0;
   virtual const G4VPhysicalVolume &GetWorldVolume() const = 0;
};
```

A concrete implementation is provided:

```
class G4WeightWindowStore: public G4VWeightWindowStore {
 public:
   explicit G4WeightWindowStore(const G4VPhysicalVolume &worldvolume);
   virtual ~G4WeightWindowStore();
   virtual G4double GetLowerWeitgh(const G4GeometryCell &gCell,
                                    G4double partEnergy) const;
   virtual G4bool IsKnown(const G4GeometryCell &gCell) const;
   virtual const G4VPhysicalVolume &GetWorldVolume() const;
   void AddLowerWeights(const G4GeometryCell &gCell,
                         const std::vector<G4double> &lowerWeights);
   void AddUpperEboundLowerWeightPairs(const G4GeometryCell &gCell,
                                        const G4UpperEnergyToLowerWeightMap&
                                        enWeMap);
   void SetGeneralUpperEnergyBounds(const
       std::set<G4double, std::less<G4double> > & enBounds);

 private::
 ...
};
```

The user may choose equal energy bounds for all cells. In this case a set of upper energy bounds must be given to the store using the method `SetGeneralUpperEnergyBounds`. If a general set of energy bounds have been set `AddLowerWeights` can be used to add the cells.

Alternatively, the user may chose different energy regions for different cells. In this case the user must provide a mapping of upper energy bounds to lower weight bounds for every cell using the method `AddUpperEboundLowerWeightPairs`.

Weight window algorithms implementing the interface class `G4VWeightWindowAlgorithm` can be used to define a customized algorithm:

```
class G4VWeightWindowAlgorithm {
 public:
   G4VWeightWindowAlgorithm();
   virtual ~G4VWeightWindowAlgorithm();
   virtual G4Nsplit_Weight Calculate(G4double init_w,
                                      G4double lowerWeightBound) const = 0;
};
```

A concrete implementation is provided and used as a default:

```
class G4WeightWindowAlgorithm : public G4VWeightWindowAlgorithm {
 public:
   G4WeightWindowAlgorithm(G4double upperLimitFaktor = 5,
                           G4double survivalFaktor = 3,
                           G4int maxNumberOfSplits = 5);
   virtual ~G4WeightWindowAlgorithm();
   virtual G4Nsplit_Weight Calculate(G4double init_w,
                                     G4double lowerWeightBound) const;
 private:
  ...
};
```

The constructor takes three parameters which are used to: calculate the upper weight bound (upperLimitFaktor), calculate the survival weight (survivalFaktor), and introduce a maximal number (maxNumberOfSplits) of copies to be created in one go.

In addition, the inverse of the maxNumberOfSplits is used to specify the minimum survival probability in case of Russian roulette.

### The Weight Roulette Technique

Weight roulette (also called weight cutoff) is usually applied if importance sampling and implicit capture are used together. Implicit capture is not described here but it is useful to note that this procedure reduces a particle weight in every collision instead of killing the particle with some probability.

Together with importance sampling the weight of a particle may become so low that it does not change any result significantly. Hence tracking a very low weight particle is a waste of computing time. Weight roulette is applied in order to solve this problem.

**The weight roulette concept**

Weight roulette takes into account the importance “Ic” of the current cell and the importance “Is” of the cell in which the source is located, by using the ratio “R=Is/Ic”.

Weight roulette uses a relative minimal weight limit and a relative survival weight. When a particle falls below the weight limit Russian roulette is applied. If the particle survives, tracking will be continued and the particle weight will be set to the survival weight.

The weight roulette uses the following parameters with their default values:

- *wsurvival*: 0.5
- *wlimit*: 0.25
- *isource*: 1

The following algorithm is applied:

If a particle weight “w” is lower than R*wlimit:

- the weight of the particle will be changed to “ws = wsurvival*R”
- the probability for the particle to survive is “p = w/ws”

## 3.7.2 Physics Based Biasing

GEANT4 supports physics based biasing through a number of general use, built in biasing techniques. A utility class, G4WrapperProcess, is also available to support user defined biasing.

### Built in Biasing Options

#### Primary Particle Biasing

Primary particle biasing can be used to increase the number of primary particles generated in a particular phase space region of interest. The weight of the primary particle is modified as appropriate. A general implementation is provided through the `G4GeneralParticleSource` class. It is possible to bias position, angular and energy distributions.

`G4GeneralParticleSource` is a concrete implementation of `G4VPrimaryGenerator`. To use, instantiate `G4GeneralParticleSource` in the `G4VUserPrimaryGeneratorAction` class, as demonstrated below.

```
MyPrimaryGeneratorAction::MyPrimaryGeneratorAction() {
   generator = new G4GeneralParticleSource;
}

void
MyPrimaryGeneratorAction::GeneratePrimaries(G4Event*anEvent){
   generator->GeneratePrimaryVertex(anEvent);
}
```

The biasing can be configured through interactive commands, as described in *General Particle Source*. Examples are also distributed with the GEANT4 distribution in **examples/extended/eventgenerator/exgps**.

#### Hadronic Leading Particle Biasing

One hadronic leading particle biasing technique is implemented in the G4HadLeadBias utility. This method keeps only the most important part of the event, as well as representative tracks of each given particle type. So the track with the highest energy as well as one of each of Baryon, pi0, mesons and leptons. As usual, appropriate weights are assigned to the particles. Setting the **SwitchLeadBiasOn** environmental variable will activate this utility.

#### Hadronic Cross Section Biasing

Cross section biasing artificially enhances/reduces the cross section of a process. This may be useful for studying thin layer interactions or thick layer shielding. The built in hadronic cross section biasing applies to photon inelastic, electron nuclear and positron nuclear processes.

The biasing is controlled through the **BiasCrossSectionByFactor** method in G4HadronicProcess, as demonstrated below.

```
void MyPhysicsList::ConstructProcess()
{
   ...
   G4ElectroNuclearReaction * theElectroReaction =
      new G4ElectroNuclearReaction;

   G4ElectronNuclearProcess theElectronNuclearProcess;
   theElectronNuclearProcess.RegisterMe(theElectroReaction);
   theElectronNuclearProcess.BiasCrossSectionByFactor(100);

   pManager->AddDiscreteProcess(&theElectronNuclearProcess);
```

(continues on next page)

(continued from previous page)

```
   ...
}
```

### Radioactive Decay Biasing

The `G4RadioactiveDecay` (GRDM) class simulates the decay of radioactive nuclei and implements the following biasing options:

- Increase the sampling rate of radionuclides within observation times through a user defined probability distribution function
- Nuclear splitting, where the parent nuclide is split into a user defined number of nuclides
- Branching ratio biasing where branching ratios are sampled with equal probability

G4RadioactiveDecay is a process which must be registered with a process manager, as demonstrated below.

```
void MyPhysicsList::ConstructProcess()
{
   ...
   G4RadioactiveDecay* theRadioactiveDecay =
      new  G4RadioactiveDecay();

   G4ProcessManager* pmanager = ...
   pmanager ->AddProcess(theRadioactiveDecay);
   ...
}
```

Biasing can be controlled either in compiled code or through interactive commands. Radioactive decay biasing examples are also distributed with the GEANT4 distribution in **examples/extended/radioactivedecay/exrdm**.

To select biasing as part of the process registration, use

```
theRadioactiveDecay->SetAnalogueMonteCarlo(false);
```

or the equivalent macro command:

```
/grdm/analogeMC [true|false]
```

In both cases, *true* specifies that the unbiased (analogue) simulation will be done, and *false* selects biasing.

### Limited Radionuclides

Radioactive decay may be restricted to only specific nuclides, in order (for example) to avoid tracking extremely long-lived daughters in decay chains which are not of experimental interest. To limit the range of nuclides decayed as part of the process registration (above), use

```
G4NucleusLimits limits(aMin, aMax, zMin, zMax);
theRadioactiveDecay->SetNucleusLimits(limits);
```

or via the macro command

```
/grdm/nucleusLimits [aMin] [aMax] [zMin] [zMax]
```

### Geometric Biasing

Radioactive decays may be generated throughout the user's detector model, in one or more specified volumes, or nowhere. The detector geometry must be defined before applying these geometric biases.

Volumes may be selected or deselected programmatically using

```
theRadioactiveDecay->SelectAllVolumes();
theRadioactiveDecay->DeselectAllVolumes();

G4LogicalVolume* aLogicalVolume;        // Acquired by the user
theRadioactiveDecay->SelectVolume(aLogicalVolume);
theRadioactiveDecay->DeselectVolume(aLogicalVolume);
```

or with the equivalent macro commands

```
/grdm/allVolumes
/grdm/noVolumes
/grdm/selectVolume [logicalVolume]
/grdm/deselectVolume [logicalVolume]
```

In macro commands, the volumes are specified by name, and found by searching the `G4LogicalVolumeStore`.

### Decay Time Biasing

The decay time function (normally an exponential in the natural lifetime) of the primary particle may be replaced with a *time profile* F(t), as discussed in Section 40.6 of the *Physics Reference Manual*. The profile function is represented as a two-column ASCII text file with up to 100 time points (first column) with fractions (second column).

```
theRadioactiveDecay->SetSourceTimeProfile(fileName);
theRadioactiveDecay->SetDecayBias(fileName);
```

```
/grdm/sourceTimeProfile [fileName]
/grdm/decayBiasProfile [fileName]
```

### Branching Fraction Biasing

Radionuclides with rare decay channels may be biased by forcing all channels to be selected uniformly (`BRBias` = *true* below), rather than according to their natural branching fractions (*false*).

```
theRadioactiveDecay->SetBRBias(true);
```

```
/grdm/BRbias [true|false]
```

### Nuclear Splitting

The statistical efficiency of generated events may be increased by generating multiple "copies" of nuclei in an event, each of which is decayed independently, with an assigned weight of 1/Nsplit. Scoring the results of tracking the decay daughters, using their corresponding weights, can improve the statistical reach of a simulation while preserving the shape of the resulting distributions.

```
theRadioactiveDecay->SetSplitNuclei(Nsplit);
```

```
/grdm/splitNucleus [Nsplit]
```

### G4WrapperProcess

G4WrapperProcess can be used to implement user defined event biasing. G4WrapperProcess, which is a process itself, wraps an existing process. By default, all function calls are forwarded to the wrapped process. It is a non-invasive way to modify the behaviour of an existing process.

To use this utility, first create a derived class inheriting from G4WrapperProcess. Override the methods whose behaviour you would like to modify, for example, PostStepDoIt, and register the derived class in place of the process to be wrapped. Finally, register the wrapped process with G4WrapperProcess. The code snippets below demonstrate its use.

```
class MyWrapperProcess  : public G4WrapperProcess {
...
 G4VParticleChange* PostStepDoIt(const G4Track& track,
                                 const G4Step& step) {
    // Do something interesting
 }
};


void MyPhysicsList::ConstructProcess()
{
...
   G4eBremsstrahlung* bremProcess =
      new G4eBremsstrahlung();

   MyWrapperProcess* wrapper = new MyWrapperProcess();
   wrapper->RegisterProcess(bremProcess);

   processManager->AddProcess(wrapper, -1, -1, 3);
}
```

## 3.7.3 Adjoint/Reverse Monte Carlo

Another powerful biasing technique available in GEANT4 is the Reverse Monte Carlo (RMC) method, also known as the Adjoint Monte Carlo method. In this method particles are generated on the external boundary of the sensitive part of the geometry and then are tracked backward in the geometry till they reach the external source surface, or exceed an energy threshold. By this way the computing time is focused only on particle tracks that are contributing to the tallies. The RMC method is much rapid than the Forward MC method when the sensitive part of the geometry is small compared to the rest of the geometry and to the external source, that has to be extensive and not beam like. At the moment the RMC method is implemented in GEANT4 only for some electromagnetic processes (see *Reverse processes*). An example illustrating the use of the Reverse MC method in GEANT4 is distributed within the GEANT4 toolkit in **examples/extended/biasing/ReverseMC01**.

### Treatment of the Reverse MC method in GEANT4

Different G4Adjoint classes have been implemented into the GEANT4 toolkit in order to run an adjoint/reverse simulation in a GEANT4 application. This implementation is illustrated in Fig. 3.3. An adjoint run is divided in a series of alternative adjoint and forward tracking of adjoint and normal particles. One GEANT4 event treats one of this tracking phase.

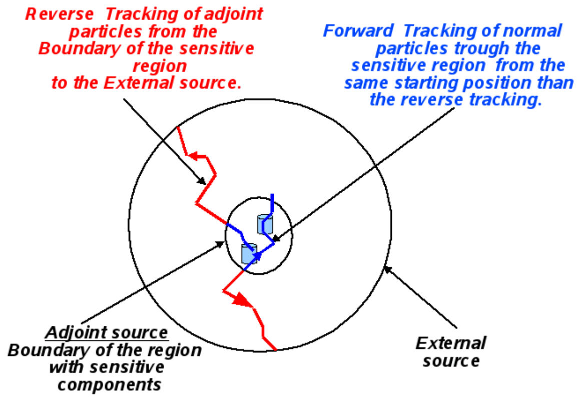

Reverse Tracking of adjoint particles from the Boundary of the sensitive region to the External source.
Forward Tracking of normal particles trough the sensitive region from the same starting position than the reverse tracking.
Adjoint source
Boundary of the region with sensitive components
External source

Fig. 3.3: Schematic view of an adjoint/reverse simulation in GEANT4.

### Adjoint tracking phase

Adjoint particles (adjoint_e-, adjoint_gamma,. . . ) are generated one by one on the so called adjoint source with random position, energy (1/E distribution) and direction. The adjoint source is the external surface of a user defined volume or of a user defined sphere. The adjoint source should contain one or several sensitive volumes and should be small compared to the entire geometry. The user can set the minimum and maximum energy of the adjoint source. After its generation the adjoint primary particle is tracked backward in the geometry till a user defined external surface (spherical or boundary of a volume) or is killed before if it reaches a user defined upper energy limit that represents the maximum energy of the external source. During the reverse tracking, reverse processes take place where the adjoint particle being tracked can be either scattered or transformed in another type of adjoint particle. During the reverse tracking the G4AdjointSimulationManager replaces the user defined primary, run, stepping, . . . actions, by its own actions. A reverse tracking phase corresponds to one GEANT4 event.

### Forward tracking phase

When an adjoint particle reaches the external surface its weight, type, position, and direction are registered and a normal primary particle, with a type equivalent to the last generated primary adjoint, is generated with the same energy, position but opposite direction and is tracked in the forward direction in the sensitive region as in a forward MC simulation. During this forward tracking phase the event, stacking, stepping, tracking actions defined by the user for his forward simulation are used. By this clear separation between adjoint and forward tracking phases, the code of the user developed for a forward simulation should be only slightly modified to adapt it for an adjoint simulation (see *How to update a G4 application to use the reverse Monte Carlo mode*). Indeed the computation of the signals is done by the same actions or classes that the one used in the forward simulation mode. A forward tracking phase corresponds to one G4 event.

### Reverse processes

During the reverse tracking, reverse processes act on the adjoint particles. The reverse processes that are at the moment available in GEANT4 are the:

- Reverse discrete ionization for e-, proton and ions
- Continuous gain of energy by ionization and bremsstrahlung for e- and by ionization for protons and ions
- Reverse discrete e- bremsstrahlung
- Reverse photo-electric effect
- Reverse Compton scattering
- Approximated multiple scattering (see comment in *Reverse multiple scattering*)

It is important to note that the electromagnetic reverse processes are cut dependent as their equivalent forward processes. The implementation of the reverse processes is based on the forward processes implemented in the G4 standard electromagnetic package.

### Nb of adjoint particle types and nb of G4 events of an adjoint simulation

The list of type of adjoint and forward particles that are generated on the adjoint source and considered in the simulation is a function of the adjoint processes declared in the physics list. For example if only the e- and gamma electromagnetic processes are considered, only adjoint e- and adjoint gamma will be considered as primaries. In this case an adjoint event will be divided in four G4 event consisting in the reverse tracking of an adjoint e-, the forward tracking of its equivalent forward e-, the reverse tracking of an adjoint gamma, and the forward tracking of its equivalent forward gamma. In this case a run of 100 adjoint events will consist into 400 GEANT4 events. If the proton ionization is also considered adjoint and forward protons are also generated as primaries and 600 GEANT4 events are processed for 100 adjoint events.

### How to update a G4 application to use the reverse Monte Carlo mode

Some modifications are needed to an existing GEANT4 application in order to adapt it for the use of the reverse simulation mode (see also the G4 example **examples/extended/biasing/ReverseMC01**). It consists into the:

- Creation of the adjoint simulation manager in the main code
- Optional declaration of user actions that will be used during the adjoint tracking phase
- Use of a special physics lists that combine the adjoint and forward processes
- Modification of the user analysis part of the code

### Creation of G4AdjointSimManager in the main

The class G4AdjointSimManager represents the manager of an adjoint simulation. This static class should be created somewhere in the main code. The way to do that is illustrated below

```
int main(int argc,char** argv) {
   ...
   G4AdjointSimManager* theAdjointSimManager = G4AdjointSimManager::GetInstance();
   ...
}
```

By doing this the G4 application can be run in the reverse MC mode as well as in the forward MC mode. It is important to note that G4AdjointSimManager is not a new G4RunManager and that the creation of G4RunManager in the main and the declaration of the geometry, physics list, and user actions to G4RunManager is still needed. The definition of the adjoint and external sources and the start of an adjoint simulation can be controlled by G4UI commands in the directory **/adjoint**.

### Optional declaration of adjoint user actions

During an adjoint simulation the user stepping, tracking, stacking and event actions declared to G4RunManager are used only during the G4 events dedicated to the forward tracking of normal particles in the sensitive region, while during the events where adjoint particles are tracked backward the following happen concerning these actions:

- The user stepping action is replaced by G4AdjointSteppingAction that is responsible to stop an adjoint track when it reaches the external source, exceed the maximum energy of the external source, or cross the adjoint source surface. If needed the user can declare its own stepping action that will be called by G4AdjointSteppingAction after the check of stopping track conditions. This stepping action can be different that the stepping action used for the forward simulation. It is declared to G4AdjointSimManager by the following lines of code:

  ```
  G4AdjointSimManager* theAdjointSimManager = G4AdjointSimManager::GetInstance();
  theAdjointSimManager->SetAdjointSteppingAction(aUserDefinedSteppingAction);
  ```

- No stacking, tracking and event actions are considered by default. If needed the user can declare to G4AdjointSimManager stacking, tracking and event actions that will be used only during the adjoint tracking phase. The following lines of code show how to declare these adjoint actions to G4AdjointSimManager:

  ```
  G4AdjointSimManager* theAdjointSimManager = G4AdjointSimManager::GetInstance();
  theAdjointSimManager->SetAdjointEventAction(aUserDefinedEventAction);
  theAdjointSimManager->SetAdjointStackingAction(aUserDefinedStackingAction);
  theAdjointSimManager->SetAdjointTrackingAction(aUserDefinedTrackingAction);
  ```

By default no user run action is considered in an adjoint simulation but if needed such action can be declared to G4AdjointSimManager as such:

```
G4AdjointSimManager* theAdjointSimManager = G4AdjointSimManager::GetInstance();
theAdjointSimManager->SetAdjointRunAction(aUserDefinedRunAction);
```

### Physics list for reverse and forward electromagnetic processes

To run an adjoint simulation a specific physics list should be used where existing G4 adjoint electromagnetic processes and their forward equivalent have to be declared. An example of such physics list is provided by the class G4AdjointPhysicsLits in the G4 example **extended/biasing/ReverseMC01**.

### Modification in the analysis part of the code

The user code should be modified to normalize the signals computed during the forward tracking phase to the weight of the last adjoint particle that reaches the external surface. This weight represents the statistical weight that the last full adjoint tracks (from the adjoint source to the external source) would have in a forward simulation. If multiplied by a signal and registered in function of energy and/or direction the simulation results will give an answer matrix of this signal. To normalize it to a given spectrum it has to be furthermore multiplied by a directional differential flux corresponding to this spectrum The weight, direction, position , kinetic energy and type of the last adjoint particle that reaches the external source, and that would represents the primary of a forward simulation, can be gotten from G4AdjointSimManager by using for example the following line of codes

```
G4AdjointSimManager* theAdjointSimManager = G4AdjointSimManager::GetInstance();
G4String particle_name = theAdjointSimManager->GetFwdParticleNameAtEndOfLastAdjointTrack();
G4int PDGEncoding= theAdjointSimManager->GetFwdParticlePDGEncodingAtEndOfLastAdjointTrack();
G4double weight = theAdjointSimManager->GetWeightAtEndOfLastAdjointTrack();
G4double Ekin  = theAdjointSimManager->GetEkinAtEndOfLastAdjointTrack();
G4double Ekin_per_nuc=theAdjointSimManager->GetEkinNucAtEndOfLastAdjointTrack(); // for ions
G4ThreeVector dir =  theAdjointSimManager->GetDirectionAtEndOfLastAdjointTrack();
G4ThreeVector pos = theAdjointSimManager->GetPositionAtEndOfLastAdjointTrack();
```

In order to have a code working for both forward and adjoint simulation mode, the extra code needed in user actions or analysis manager for the adjoint simulation mode can be separated to the code needed only for the normal forward simulation by using the following public method of G4AdjointSimManager:

```
G4bool GetAdjointSimMode();
```

that returns true if an adjoint simulation is running and false if not.

The following code example shows how to normalize a detector signal and compute an answer matrix in the case of an adjoint simulation.

Listing 3.5: Normalization in the case of an adjoint simulation. The detector signal S computed during the forward tracking phase is normalized to a primary source of e- with a differential directional flux given by the function F. An answer matrix of the signal is also computed.

```
G4double S = ...; // signal in the sensitive volume computed during a forward tracking phase

//Normalization of the signal for an adjoint simulation
G4AdjointSimManager* theAdjSimManager = G4AdjointSimManager::GetInstance();
if (theAdjSimManager->GetAdjointSimMode()) {
  G4double normalized_S=0.;         //normalized to a given e- primary spectrum
  G4double S_for_answer_matrix=0.; //for e- answer matrix

  if (theAdjSimManager->GetFwdParticleNameAtEndOfLastAdjointTrack() == "e-") {
    G4double ekin_prim = theAdjSimManager->GetEkinAtEndOfLastAdjointTrack();
    G4ThreeVector dir_prim =  theAdjointSimManager->GetDirectionAtEndOfLastAdjointTrack();
    G4double weight_prim = theAdjSimManager->GetWeightAtEndOfLastAdjointTrack();
    S_for_answer_matrix = S*weight_prim;
    normalized_S = S_for_answer_matrix*F(ekin_prim,dir);
    // F(ekin_prim,dir_prim) gives the differential directional flux of primary e-
  }
  //follows the code where normalized_S and S_for_answer_matrix are registered or whatever
   ....
}

//analysis/normalization code for forward simulation
else {
  ....
}
....
```

## Control of an adjoint simulation

The G4UI commands in the directory /adjoint. allow the user to :

- Define the adjoint source where adjoint primaries are generated
- Define the external source till which adjoint particles are tracked
- Start an adjoint simulation

#### Known issues in the Reverse MC mode

#### Occasional wrong high weight in the adjoint simulation

In rare cases an adjoint track may get a wrong high weight when reaching the external source. While this happens not often it may corrupt the simulation results significantly. This happens in some tracks where both reverse photo-electric and bremsstrahlung processes take place at low energy. We still need some investigations to remove this problem at the level of physical adjoint/reverse processes. However this problem can be solved at the level of event actions or analysis in the user code by adding a test on the normalized signal during an adjoint simulation. An example of such test has been implemented in the GEANT4 example **extended/biasing/ReverseMC01**. In this implementation an event is rejected when the relative error of the computed normalized energy deposited increases during one event by more than 50% while the computed precision is already below 10%.

#### Reverse bremsstrahlung

A difference between the differential cross sections used in the adjoint and forward bremsstrahlung models is the source of a higher flux of >100 keV gamma in the reverse simulation compared to the forward simulation mode. In principle the adjoint processes/models should make use of the direct differential cross section to sample the adjoint secondaries and compute the adjoint cross section. However due to the way the effective differential cross section is considered in the forward model G4eBremsstrahlungModel this was not possible to achieve for the reverse bremsstrahlung. Indeed the differential cross section used in G4AdjointeBremstrahlungModel is obtained by the numerical derivation over the cut energy of the direct cross section provided by G4eBremsstrahlungModel. This would be a correct procedure if the distribution of secondary in G4eBremsstrahlungModel would match this differential cross section. Unfortunately it is not the case as independent parameterization are used in G4eBremsstrahlungModel for both the cross sections and the sampling of secondaries. (It means that in the forward case if one would integrate the effective differential cross section considered in the simulation we would not find back the used cross section). In the future we plan to correct this problem by using an extra weight correction factor after the occurrence of a reverse bremsstrahlung. This weight factor should be the ratio between the differential CS used in the adjoint simulation and the one effectively used in the forward processes. As it is impossible to have a simple and direct access to the forward differential CS in G4eBremsstrahlungModel we are investigating the feasibility to use the differential CS considered in G4Penelope models.

#### Reverse multiple scattering

For the reverse multiple scattering the same model is used than in the forward case. This approximation makes that the discrepancy between the adjoint and forward simulation cases can get to a level of ~ 10-15% relative differences in the test cases that we have considered. In the future we plan to improve the adjoint multiple scattering models by forcing the computation of multiple scattering effect at the end of an adjoint step.

### 3.7.4 Generic Biasing

The generic biasing scheme provides facilities for:

- physics-based biasing, to alter the behavior of existing physics processes:
  - biasing of physics process interaction occurrence,
  - biasing of physics process final state production;
- non-physics-based biasing, to introduce or remove particles in the simulation but without affecting the existing physics processes, with techniques like, but not limited to
  - splitting,
  - Russian roulette (killing).

Decisions on what techniques to apply are taken on a step by step and intra-step basis, hence providing a lot of flexibility.

The scheme has been introduced in 10.0, with new features and some non-backward compatible changes introduced in 10.1 and 10.2; these are documented in *Changes from 10.0 to 10.1* and *Changes from 10.1 to 10.2*. Parallel geometry capability has been introduced in 10.3.

## Overview

The generic biasing scheme relies on two abstract classes, that are meant to model the biasing problems. You have to inherit from them to create your own concrete classes, or use some of the concrete instances provided (see *Existing biasing operations, operator and interaction laws*), if they respond to your case. A dedicated process provides the interface between these biasing classes and the tracking. In case of parallel geometry usage, an other process handles the navigation in these geometries.

The two abstract classes are:

- `G4VBiasingOperation`: which represents a simple, or "atomic" biasing operation, like changing a process interaction occurrence probability, or changing its final state production, or making a splitting operation, etc. For the occurrence biasing case, the biasing is handled with an other class, **``G4VBiasingInteractionLaw``**, which holds the properties of the biased interaction law. An object of this class type must be provided by the occurrence biasing operation returned.
- `G4VBiasingOperator`: which purpose is to make decisions on the above biasing operations to be applied. It is attached to a G4LogicalVolume and is the pilot of the biasing in this volume. An operator may decide to delegate to other operators. An operator acts only in the `G4LogicalVolume` it is attached to. In volumes with no biasing operator attached, the usual tracking is applied.

The process acting as interface between the biasing classes and the tracking is:

- `G4BiasingProcessInterface`: it is a concrete `G4VProcess` implementation. It interrogates the current biasing operator, if any, for biasing operations to be applied. The `G4BiasingProcessInterface` can either:
  - hold a physics process that it wraps and controls: in this case it asks the operator for physics-based biasing operations (only) to be applied to the wrapped process,
  - not hold a physics process: in this case it asks the operator for non-physics-based biasing operations (only): splitting, killing, etc.
- The `G4BiasingProcessInterface` class provides many information that can be used by the biasing operator. Each `G4BiasingProcessInterface` provides its identity to the biasing operator it calls, so that the operator has this information but also information of the underneath wrapped physics process, if it is the case.
  The `G4BiasingProcessInterface` can be asked for all other `G4BiasingProcessInterface` instances at play on the current track. In particular, this allows the operator to get all cross-sections at the current point (feature available since 10.1). The code is organized in such a way that these cross-sections are all available at the first call to the operator in the current step.
- To make `G4BiasingProcessInterface` instances wrapping physics processes, or to insert instances not holding a physics process, the physics list has to be modified -the generic biasing approach is hence invasive to the physics list-. The way to configure your physics list and related helper tools are described below.

The process handling parallel geometries is:

- `G4ParallelGeometriesLimiterProcess`, it is a concrete `G4VProcess` implementation, which takes care of limiting the step on the boundaries of parallel geometries.
- A single instance of `G4ParallelGeometriesLimiterProcess` handles all parallel geometries to be considered for a particle type. It collects these geometries by means of `myLimiterProcess->AddParallelWorld("myParallelGeometry")` calls.
  Given such a process is attached to a particle type, parallel geometries are hence specified per particle type.

- Attaching an instance of this process to a given particle type, and specifying the parallel geometries to be considered is eased by the helper tools as explained below.

### Getting Started

### Examples

Seven "Generic Biasing (GB)" examples are proposed (they have been introduced in 10.0, 10.1, 10.3 and 10.6):

- `examples/extended/biasing/GB01`:
  - which shows how biasing of process cross-section can be done.
  - This example uses the physics-based biasing operation `G4BOptnChangeCrossSection` defined in `geant4/source/processes/biasing/generic`. This operation performs the actual process cross-section change. In the example a first `G4VBiasingOperator`, `GB01BOptrChangeCrossSection`, configures and selects this operation. This operator applies to only one particle type.
  - To allow several particle types to be biased, a second `G4VBiasingOperator`, `GB01BOptrMultiParticleChangeCrossSection`, is implemented, and which holds a `GB01BOptrChangeCrossSection` operator for each particle type to be biased. This second operator then delegates to the first one the handling of the biasing operations.
- `examples/extended/biasing/GB02`:
  - which shows how a "force collision" scheme very close to the MCNP one can be activated.
  - This second example has a quite similar approach than the `GB01` one, with a `G4VBiasingOperator`, `QGB02BOptrMultiParticleForceCollision`, that holds as many operators than particle types to be biased, this operators being of `G4BOptrForceCollision` type.
  - This `G4BOptrForceCollision` operator is defined in `source/processes/biasing/generic`. It combines several biasing operations to build-up the needed logic (see *Setting up the application*). It can be in particular looked at to see how it collects and makes use of physics process cross-sections.
- `examples/extended/biasing/GB03`:
  - which implements a kind of importance geometry biasing, using the generic biasing classes.
  - The example uses a simple sampling calorimeter. On the boundary of the absorber parts, it does splitting (killing) if the track is moving forward (backward). As the splitting can be too strong in some cases, falling into an over-splitting situation, even with a splitting by a factor 2, a technique is introduced to alleviate the problem : a probability to apply the splitting (killing) is introduced, and with proper tuning of this probability, the over-splitting can be avoided.
- `examples/extended/biasing/GB04`:
  - which implements a bremsstrahlung splitting. Bremsstrahlung splitting exists in the EM package. In the present example, it is shown how to implement a similar technique, using the generic biasing classes.
  - A biasing operator, `GB04BOptrBremSplitting`, sends a final state biasing operation, `GB04BOptnBremSplitting`, for the bremsstrahlung process. Splitting factor, and options to control the biasing are available through command line.
- `examples/extended/biasing/GB05`:
  - which illustrates a technique that uses physics cross-sections to determine the splitting[killing] rate in a shielding problem, it is applied to neutrons. This technique is supposed to be an invention, to illustrate a technique combining physics-based information with splitting/killing.
  - In the classical treatment of the shielding problem, the shield is divided in slices at the boundaries of which particles are splitted[killed] if moving forward[backward]. In the present technique, we collect the cross-sections of "absorbing/destroying" processes : decay, capture, inelastic. We then use the generic biasing facilities to create an equivalent of a splitting process, that has a "cross-section" which is the sum of the previous ones. This process is competing with other processes, as a regular one. When this process wins the competition, it splits the track, with a splitting factor 2. This splitting is hence occurring at the same rate than the absorption, resulting in an expected maintained (unweighted) flux.

- `GB05BOptrSplitAndKillByCrossSection` and `GB05BOptnSplitAndKillByCrossSection` are respectively the biasing operator and operation. The operator collects the absorbing cross-sections at the beginning of the step, passes them to the operation, requests it to sample the distance to its next interaction, and returns this operation to the calling `G4BiasingProcessInterface` as the operation to be applied in the step.
- The operation interaction distance is then proposed by the calling `G4BiasingProcessInterface` and, if being the shortest of the interaction distances, the operation final state generation (the splitting) is applied by the process.

• `examples/extended/biasing/GB06`:
  - which demonstrates the use of parallel geometries in generic biasing, on a classical shield problem, using geometry-based importance biasing.
  - The mass geometry consists of a single block of concrete; it is overlayed by a parallel geometry defining the slices used for splitting/killing.
  - The navigation capability in the parallel geometry is activated in the main program, by means of the physics list constructor.

• `examples/extended/biasing/GB07`:
  - which demonstrates the use of the leading particle biasing technique in generic biasing.
  - The mass geometry consists of a block of concrete in which the biasing is applied. A thin volume then follows to score (simple printing) the particles leaving the block of concrete.

### Setting up the application

For making an existing `G4VBiasingOperator` used by your application, you have to do two things:

1. Attach the operator to the `G4LogicalVolume` where the biasing should take place: You have to make this attachment in your `ConstructSDandField()` method (to make your application both sequential and MT-compliant):

   Listing 3.6: Attachment of a `G4BiasingOperator` to a `G4LogicalVolume`. We assume such a volume has been created with name "volumeWithBiasing", and we assume that a biasing operator class `MyBiasingOperator` has been created, inheriting from `G4VBiasingOperator`:

```
// Fetch the logical volume pointer by name (it is an example, not a mandatory way):
G4LogicalVolume* biasingVolume = G4LogicalVolumeStore::GetInstance()->GetVolume(
↪"volumeWithBiasing");
// Create the biasing operator:
MyBiasingOperator* myBiasingOperator = new MyBiasingOperator("ExampleOperator");
// Attach it to the volume:
myBiasingOperator->AttachTo(biasingVolume);
```

2. Setup the physics list you use to properly include the needed `G4BiasingProcessInterface` instances. You have several options for this.
   - The easiest way is if you use a pre-packaged physics list (e.g. `FTFP_BERT`, `QGSP...`). As such a physics list is of `G4VModularPhysicsList` type, you can alter it with a `G4VPhysicsConstructor`. The constructor `G4GenericBiasingPhysics` is meant for this. It can be used, typically in your main program, as:

Listing 3.7: Use of the `G4GenericBiasingPhysics` physics constructor to setup a pre-packaged physics list (of `G4VModularPhysicsList` type). Here we assume the `FTFP_BERT` physics list, and we assume that `runManager` is a pointer on a created `G4RunManager` or `G4RMTunManager` object.

```
// Instantiate the physics list:
FTFP_BERT* physicsList = new FTFP_BERT;
// Create the physics constructor for biasing:
G4GenericBiasingPhysics* biasingPhysics = new G4GenericBiasingPhysics();
// Tell what particle types have to be biased:
biasingPhysics->Bias("gamma");
biasingPhysics->Bias("neutron");
// Register the physics constructor to the physics list:
physicsList->RegisterPhysics(biasingPhysics);
// Set this physics list to the run manager:
runManager->SetUserInitialization(physicsList);
```

Doing so, all physics processes will be wrapped, and, for example, the gamma conversion process, `"conv"`, will appear as `"biasWrapper(conv)"` when dumping the processes (`/particle/process/dump`). An additional `"biasWrapper(0)"` process, for non-physics-based biasing is also inserted.
Other methods to specifically chose some physics processes to be biased or to insert only `G4BiasingProcessInterface` instances for non-physics-based biasing also exist.

- The second way is useful if you write your own physics list, and if this one is not a modular physics list, but inherits directly from the lowest level abstract class `G4VUserPhysicsList`. In this case, the above solution with `G4GenericBiasingPhysics` does not apply. Instead you can use the `G4BiasingHelper` utility class (this one is indeed used by `G4GenericBiasingPhysics`).

Listing 3.8: Use of the `G4BiasingHelper` utility class to setup a physics list for biasing in case this physics list is not of `G4VModularPhysicsList` type but inherits directly from `G4VUserPhysicsList`.

```
// Get physics list helper:
G4PhysicsListHelper* ph = G4PhysicsListHelper::GetPhysicsListHelper();
...
// Assume "particle" is a pointer on a G4ParticleDefinition object
G4String particleName = particle->GetParticleName();
if (particleName == "gamma")
{
ph->RegisterProcess(new G4PhotoElectricEffect , particle);
ph->RegisterProcess(new G4ComptonScattering ,   particle);
ph->RegisterProcess(new G4GammaConversion ,     particle);
G4ProcessManager* pmanager = particle->GetProcessManager();
G4BiasingHelper::ActivatePhysicsBiasing(pmanager, "phot");
G4BiasingHelper::ActivatePhysicsBiasing(pmanager, "compt");
G4BiasingHelper::ActivatePhysicsBiasing(pmanager, "conv");
G4BiasingHelper::ActivateNonPhysicsBiasing(pmanager);
}
```

- A last way to setup the physics list is by direct insertion of the `G4BiasingProcessInterface` instances, but this requires solid expertise in physics list creation.

In case you also use parallel geometries, you have to make the generic biasing sensitive to these. Assuming you have created three parallel geometries with names `"parallelWorld1"`, `"parallelWorld2"` and `"parallelWorld3"` that you want to be active for neutrons, the additional calls you have to make compared to example *EvtBias.GenericBiasing.Overview.UsePhysConstructor* above are simply:

Listing 3.9: Calls to activate parallel geometry navigation

```
// -- activate parallel geometries for neutrons:
biasingPhysics->AddParallelGeometry("neutron","parallelWorld1");
biasingPhysics->AddParallelGeometry("neutron","parallelWorld2");
biasingPhysics->AddParallelGeometry("neutron","parallelWorld3");
```

It is also possible, even though less convenient, to use the `G4BiasingHelper` utility class making calls to the static method `limiter = G4BiasingHelper::AddLimiterProcess(pmanager, "limiterProcessName")` in addition to the ones of example *EvtBias.GenericBiasing.Overview.UseBiasingHelper* above. This call returns a pointer `limiter` on the constructed `G4ParallelGeometriesLimiterProcess` process, setting its name as `"limiterProcessName"`, this pointer has then to be used to specify the parallel geometries to the process : `limiter->AddParallelWorld("parallelWorld1")...`

### Existing biasing operations, operator and interaction laws

Below are the set of available concrete biasing operations, operators and interaction laws. These are defined in `source/processes/biasing/generic`. Please note that several examples (*Examples*) also implement dedicated operators and operations.

- Concrete implementation classes of `G4VBiasingOperation`:
  - `G4BOptnCloning`: a non-physics-based biasing operation that clones the current track. Each of the two copies is given freely a weight.
  - `G4BOptnChangeCrossSection`: a physics-based biasing operation to change one process cross-section
  - `G4BOptnForceFreeFlight`: a physics-based biasing operation to force a flight with no interaction through the current volume. This operation is better said a "silent flight": the flight is conducted under a zero weight, and the track weight is restored at the end of the free flight, taking into account the cumulative weight change for the non-interaction flight. This special feature is because this class in used in the MCNP-like force collision scheme `G4BOptrForceCollision`.
  - `G4BOptnForceCommonTruncatedExp`: a physics-based biasing operation to force a collision inside the current volume. It is "common" as several processes may be forced together, driving the related interaction law by the sum of these processes cross-section. The relative natural occurrence of processes is conserved. This operation makes use of a "truncated exponential" law, which is the exponential law limited to a segment [0,L], where L is the distance to exit the current volume.
  - `G4BOptnLeadingParticle`: a non-physics-based biasing operation that implements a Leading Particle Biasing scheme. The technique can be applied to hadronic, electromagnetic et decay processes. At each interaction point, are kept:
    - the leading particle (highest energy track),
    - one particle of each species (considering particles and anti-particles as of same species, and all particles with Z >= 2 as one species).

    A Russian roulette is additionnally played on the surviving non-leading tracks. This is specially of interest for electromagnetic processes as these have low multiplicities, making unaffected the final state if applying the above algorithm. The default killing probability is 2/3, but can be changed by the `void SetFurtherKillingProbability(G4double p)` method.
- Concrete implementation class of `G4VBiasingOperator`:
  - `G4BOptrForceCollision`: a biasing operator that implements a force collision scheme quite close to the one provided by MCNP. It handles the scheme though the following sequence:
    1. The operator starts by using a `G4BOptnCloning` cloning operation, making a copy of the primary entering the volume. The primary is given a zero weight.
    2. The primary is then transported through to the volume, without interactions. This is done with the operator requesting forced free flight `G4BOptnForceFreeFlight` operations to all physics processes. The weight is zero to prevent the primary to contribute to scores. This flight purpose is to accumulate the probability to fly through the volume without interaction. When the primary reaches

the volume boundary, the first free flight operation restores the primary weight to its initial weight and all operations multiply this weight by their weight for non-interaction flight. The operator then abandons here the primary track, letting it back to normal tracking.

3. The copy of the primary track starts and the track is forced to interact in the volume, using the `G4BOptnForceCommonTruncatedExp` operation, itself using the total cross-section to compute the forced interaction law (exponential law limited to path length in the volume). One of the physics processes is randomly selected (on the basis of cross-section values) for the interaction.
4. Other processes are receiving a forced free flight operation, from the operator.
5. The copy of the primary is transported up to its interaction point. With these operations configured, the `G4BiasingProcessInterface` instances have all needed information to automatically compute the weight of the primary track and of its interaction products.

As this operation starts on the volume boundary, a single force interaction occurs: if the track survives the interaction (e.g Compton process), as it moved apart the boundary, the operator does not consider it further.

- `G4VBiasingInteractionLaw` classes. These classes describe the interaction law in term of a non-interaction probability over a segment of length l, and an "effective" cross-section for an interaction at distance l (see Physics Reference Manual, section generic biasing). An interaction law can also be sampled.
  - `G4InteractionLawPhysical`: the usual exponential law, driven by a cross-section constant over a step. The effective cross-section is the cross-section.
  - `G4ILawForceFreeFlight`: an "interaction" law for, precisely, a non-interacting track, with non-interaction probability always 1, and zero effective cross-section. It is a limit case of the modeling.
  - `G4ILawTruncatedExp`: an exponential interaction law limited to a segment [0,L]. The non-interaction probability and effective cross-section depend on l, the distance travelled, and become zero and infinite, respectively, at l=L.

## Changes from 10.0 to 10.1

The `G4VBiasingOperation` class has been evolved to simplify the interface. The changes regard physics-based biasing (occurrence biasing and final state biasing) and are:

- Suppression of the method `virtual G4ForceCondition ProposeForceCondition(const G4ForceCondition wrappedProcessCondition)`
  - The functionality has been kept, absorbing the `ProposeForceCondition(...)` method by the `ProvideOccurenceBiasingInteractionLaw(...)` one, which has now the signature:

```
virtual const G4VBiasingInteractionLaw*
ProvideOccurenceBiasingInteractionLaw ( const
G4BiasingProcessInterface* callingProcess, G4ForceCondition&
proposeForceCondition) = 0;
```

  - The value of `proposeForceCondition` passed to the method is the `G4ForceCondition` value of the wrapped process, as this was the case with deprecated method `ProposeForceCondition(...)` .
- Suppression of the virtual method "G4bool DenyProcessPostStepDoIt(const G4BiasingProcessInterface* callingProcess, const G4Track* track, const G4Step* step, G4double& proposedTrackWeight)":
  - This method was used to prevent the wrapped process hold by `callingProcess` to have its `PostStepDoIt(...)` called, providing a weight for this non-call.
  - The method has been removed, but the functionality still exists, and has been merged and generalized with the change of the pure virtual `ApplyFinalStateBiasing(...)` described just after.
- Extra argument `G4bool& forceBiasedFinalState` added as last argument of `virtual G4VParticleChange* ApplyFinalStateBiasing( const G4BiasingProcessInterface* callingProcess, const G4Track* track, const G4Step* step, G4bool& forceBiasedFinalState) = 0`
  - This method is meant to return a final state interaction through the `G4VParticleChange`. The final state may be the analog wrapped process one, or a biased one, which comes with its weight correction

for biasing the final state. If an occurrence biasing is also at play in the same step, the weight correction for this biasing is applied to the final state before this one is returned to the stepping. This is the default behavior. This behavior can be controlled by forceBiasedFinalState:

  * If `forceBiasedFinalState` is left `false`, the above default behavior is applied.
  * If `forceBiasedFinalState` is set to `true`, the `G4VParticleChange` final state will be *returned as is* to the stepping, and that, *regardless* there is an occurrence at play. Hence, when setting `forceBiasedFinalState` to `true`, the biasing operation *takes full responsibility* for the total weight (occurrence + final state) calculation.

- Deletion of `G4ILawCommonTruncatedExp`, which could be eliminated after better implementation of `G4BOptnForceCommonTruncatedExp` operation.

### Changes from 10.1 to 10.2

Changes in 10.2 derive from the introduction of the `track` feature `G4VAuxiliaryTrackInformation`. They regard essentially the force collision operator `G4BOptrForceCollision` and related features. These changes are transparent to the user if using `G4BOptrForceCollision` and following `examples/extended/biasing/GB02`. The information below are provided for developers of biasing classes.

The `G4VAuxiliaryTrackInformation` functionality allows to extend the `G4Track` attributes with an instance of a concrete class deriving from `G4VAuxiliaryTrackInformation`. Such an object is registered to the `G4Track` using an `ID` that has to be previously obtained from the `G4PhysicsModelCatalog`. The `G4VBiasingOperator` class defines two new virtual methods, `Configure()` and `ConfigureForWorker()`, to help with the creation of these `ID's` at the proper time (see `G4BOptrForceCollision` as an example).

Before 10.2, the `G4BOptrForceCollision` class was using state variables to make the bookkeeping of the tracks handled by the scheme. Now this bookkeeping is handled using a `G4VAuxiliaryTrackInformation`, `G4BOptrForceCollisionTrackData`.

To help with the bookkeeping, the base class `G4VBiasingOperator` was defining a set of methods (`GetBirthOperation(..)`, `RememberSecondaries(..)`, `ForgetTrack(..)`), these have been removed in 10.2 and are easy to overpass with a dedicated `G4VAuxiliaryTrackInformation`.

CHAPTER

# FOUR

# DETECTOR DEFINITION AND RESPONSE

## 4.1 Geometry

### 4.1.1 Introduction

The detector definition requires the representation of its geometrical elements, their materials and electronics properties, together with visualization attributes and user defined properties. The geometrical representation of detector elements focuses on the definition of solid models and their spatial position, as well as their logical relations to one another, such as in the case of containment.

GEANT4 uses the concept of "Logical Volume" to manage the representation of detector element properties. The concept of "Physical Volume" is used to manage the representation of the spatial positioning of detector elements and their logical relations. The concept of "Solid" is used to manage the representation of the detector element solid modeling. Volumes and solids must be dynamically allocated using 'new' in the user program; they must not be declared as local objects. Volumes and solids are automatically registered on creation to dedicated stores; these stores will delete all objects at the end of the job.

### 4.1.2 Solids

The GEANT4 geometry modeller implements Constructive Solid Geometry (CSG) representations for geometrical primitives. CSG representations are easy to use and normally give superior performance.

All solids must be allocated using 'new' in the user's program; they get registered to a `G4SolidStore` at construction, which will also take care to deallocate them at the end of the job, if not done already in the user's code.

All constructed solids can stream out their contents via appropriate methods and streaming operators.

For all solids it is possible to estimate the geometrical volume and the surface area by invoking the methods:

```
G4double GetCubicVolume()
G4double GetSurfaceArea()
```

which return an estimate of the solid volume and total area in internal units respectively. For elementary solids the functions compute the exact geometrical quantities, while for composite or complex solids an estimate is made using Monte Carlo techniques.

For all solids it is also possible to generate pseudo-random points lying on their surfaces, by invoking the method

```
G4ThreeVector GetPointOnSurface() const
```

which returns the generated point in local coordinates relative to the solid. To be noted that this function is not meant to provide a uniform distribution of points on the surfaces of the solids.

Since release 10.3, solids can be scaled in their dimensions along the Cartesian axes X, Y or Z, by providing a scale transformation associated to the original solid.

```
G4ScaledSolid( const G4String& pName,
                     G4VSolid* pSolid ,
               const G4Scale3D& pScale  )
```

**Note:** GEANT4 does not impose any restriction on the name assigned to solids; names can be shared. It is however good practice to specify unique names for each constructed solid, to allow for easier retrivial from stores for post-processing use.

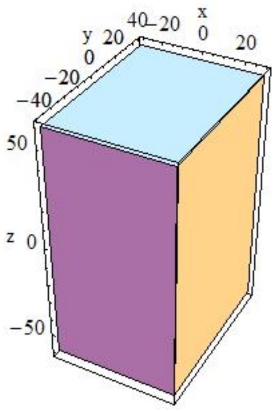
### Constructed Solid Geometry (CSG) Solids

CSG solids are defined directly as three-dimensional primitives. They are described by a minimal set of parameters necessary to define the shape and size of the solid. CSG solids are Boxes, Tubes and their sections, Cones and their sections, Spheres, Wedges, and Toruses.

**Box:**

To create a **box** one can use the constructor:

```
G4Box(const G4String& pName,
             G4double  pX,
             G4double  pY,
             G4double  pZ)
```



*In the picture*:
`pX = 30, pY = 40, pZ = 60`

by giving the box a name and its half-lengths along the X, Y and Z axis:

| | | | | | |
|---|---|---|---|---|---|
| pX | half length in X | pY | half length in Y | pZ | half length in Z |

This will create a box that extends from -pX to +pX in X, from -pY to +pY in Y, and from -pZ to +pZ in Z.

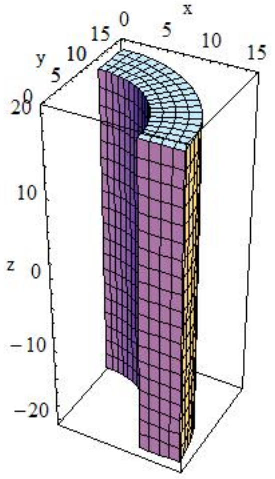
For example to create a box that is 2 by 6 by 10 centimeters in full length, and called BoxA one should use the following code:

```
G4Box* aBox = new G4Box("BoxA", 1.0*cm, 3.0*cm, 5.0*cm);
```

**Cylindrical Section or Tube:**

Similarly to create a **cylindrical section** or **tube**, one would use the constructor:

```
G4Tubs(const G4String& pName,
             G4double  pRMin,
             G4double  pRMax,
             G4double  pDz,
             G4double  pSPhi,
             G4double  pDPhi)
```



*In the picture*:
`pRMin = 10, pRMax = 15, pDz = 20`

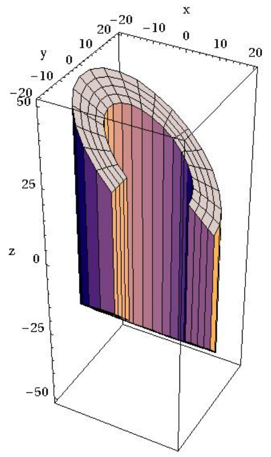
giving its name `pName` and its parameters which are:

| | | | |
|---|---|---|---|
| `pRMin` | Inner radius | `pRMax` | Outer radius |
| `pDz` | Half length in Z | `pSPhi` | Starting phi angle in radians |
| `pDPhi` | Angle of the segment in radians | | |

**Cylindrical Cut Section or Cut Tube:**

A cut in `Z` can be applied to a cylindrical section to obtain a **cut tube**. The following constructor should be used:

```
G4CutTubs(const G4String& pName,
                G4double pRMin,
                G4double pRMax,
                G4double pDz,
                G4double pSPhi,
                G4double pDPhi,
                G4ThreeVector pLowNorm,
                G4ThreeVector pHighNorm)
```



*In the picture*:
`pRMin = 12, pRMax = 20, pDz = 30, pSPhi = 0, pDPhi = 1.5*pi, pLowNorm = (0,-0.7,-0.71), pHighNorm = (0.7,0,0.71)`

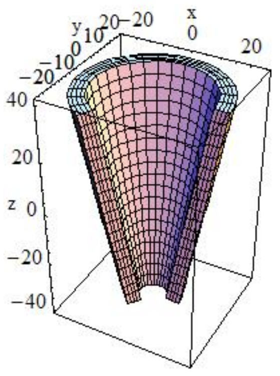
giving its name `pName` and its parameters which are:

| | | | |
|---|---|---|---|
| pRMin | Inner radius | pRMax | Outer radius |
| pDz | Half length in Z | pSPhi | Starting phi angle in radians |
| pDPhi | Angle of the segment in radians | pLowNorm | Outside Normal at -Z |
| pHighNorm | Outside Normal at +Z | | |

**Cone or Conical section:**

Similarly to create a **cone**, or **conical section**, one would use the constructor

```
G4Cons(const G4String& pName,
              G4double  pRmin1,
              G4double  pRmax1,
              G4double  pRmin2,
              G4double  pRmax2,
              G4double  pDz,
              G4double  pSPhi,
              G4double  pDPhi)
```



*In the picture*: pRmin1 = 5, pRmax1 = 10, pRmin2 = 20, pRmax2 = 25, pDz = 40, pSPhi = 0, pDPhi = 4/3*Pi

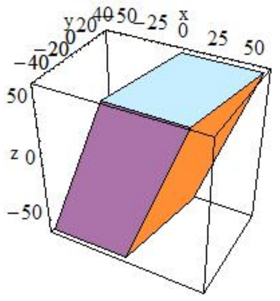
giving its name pName, and its parameters which are:

| | | | |
|---|---|---|---|
| pRmin1 | inside radius at −pDz | pRmax1 | outside radius at −pDz |
| pRmin2 | inside radius at +pDz | pRmax2 | outside radius at +pDz |
| pDz | half length in Z | pSPhi | starting angle of the segment in radians |
| pDPhi | the angle of the segment in radians | | |

**Parallelepiped:**

A **parallelepiped** is constructed using:

```
G4Para(const G4String& pName,
              G4double  dx,
              G4double  dy,
              G4double  dz,
              G4double  alpha,
              G4double  theta,
              G4double  phi)
```



*In the picture*: dx = 30, dy = 40, dz = 60

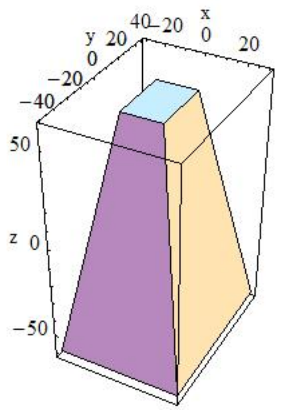
giving its name pName and its parameters which are:

| | |
|---|---|
| dx,dy, dz | Half-length in x,y,z |
| alpha | Angle formed by the Y axis and by the plane joining the centre of the faces *parallel* to the Z-X plane at -dy and +dy |
| theta | Polar angle of the line joining the centres of the faces at -dz and +dz in Z |
| phi | Azimuthal angle of the line joining the centres of the faces at -dz and +dz in Z |

**Trapezoid:**

To construct a **trapezoid** use:

```
G4Trd(const G4String& pName,
            G4double  dx1,
            G4double  dx2,
            G4double  dy1,
            G4double  dy2,
            G4double  dz)
```



*In the picture*:
dx1 = 30, dx2 = 10, dy1 = 40, dy2 = 15, dz = 60

to obtain a solid with name pName and parameters

| | |
|---|---|
| dx1 | Half-length along X at the surface positioned at −dz |
| dx2 | Half-length along X at the surface positioned at +dz |
| dy1 | Half-length along Y at the surface positioned at −dz |
| dy2 | Half-length along Y at the surface positioned at +dz |
| dz | Half-length along Z axis |

**Generic Trapezoid:**

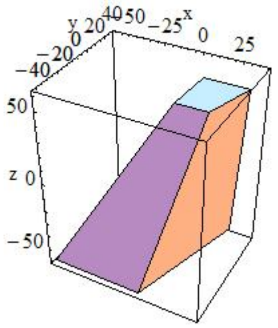
To build a **generic trapezoid**, the G4Trap class is provided. G4Trap is a solid with six trapezoidal faces, it has two bases parallel to the XY-plane and four lateral faces. The bases are located at the same distance from the XY-plane, but on opposite sides from it. Each of the bases has two edges parallel the X-axis. Let's call the line joining middle point of these edges - *the centre line of the base*, and the middle point of this line - *the centre of the base*. An important property of G4Trap is that the line joining the centres of the bases goes through the origin of the local coordinate system.

G4Trap has three main constructors; for a Right Angular Wedge, for a general trapezoid and a constructor from eight points:

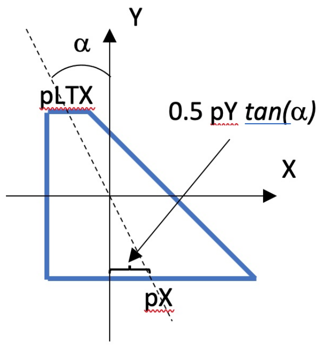
```
G4Trap(const G4String& pName,
             G4double  pZ,
             G4double  pY,
             G4double  pX,
             G4double  pLTX)

G4Trap(const G4String& pName,
             G4double  pDz,   G4double␣
↪pTheta,
             G4double  pPhi,  G4double pDy1,
             G4double  pDx1,  G4double pDx2,
             G4double  pAlp1, G4double pDy2,
             G4double  pDx3,  G4double pDx4,
             G4double  pAlp2)

G4Trap(const G4String& pName,
       const G4ThreeVector pt[8])
```



*In the picture*: `pDx1 = 30, pDx2 = 40, pDy1 = 40, pDx3 = 10, pDx4 = 14, pDy2 = 16, pDz = 60, pTheta = 20*Degree, pPhi = 5*Degree, pAlp1 = pAlp2 = 10*Degree`



to obtain a Right Angular Wedge with name `pName` and parameters:

| | |
|---|---|
| `pZ` | Length along Z |
| `pY` | Length along Y |
| `pX` | Length along X at the wider side |
| `pLTX` | Length along X at the narrower side (`plTX<=pX`) |

The angle between the Y-axis and the centre lines of the bases in case of Right Angular Wedge is defined by the following expression:

tan(alpha) = 0.5 * (pLTX - pX) / pY

or, to obtain the general trapezoid:

| | |
|---|---|
| `pDz` | Half Z length - distance from the origin to the bases |
| `pTheta` | Polar angle of the line joining the centres of the bases at -/+pDz |
| `pPhi` | Azimuthal angle of the line joining the centre of the base at -pDz to the centre of the base at +pDz |
| `pDy1` | Half Y length of the base at -pDz |
| `pDy2` | Half Y length of the base at +pDz |
| `pDx1` | Half X length at smaller Y of the base at -pDz |
| `pDx2` | Half X length at bigger Y of the base at -pDz |
| `pDx3` | Half X length at smaller Y of the base at +pDz |
| `pDx4` | Half X length at bigger y of the base at +pDz |
| `pAlp1` | Angle between the Y-axis and the centre line of the base at -pDz (lower endcap) |
| `pAlp2` | Angle between the Y-axis and the centre line of the base at +pDz (upper endcap) |

**Note:** The angle `pAlph1` and `pAlph2` have to be the same due to the planarity condition.

or, to obtain from eight points with name `pName`:

| `pt` | Coordinates of the vertices |
|---|---|
| `pt[0],pt[1]` | Edge with smaller Y of the base at -z |
| `pt[2],pt[3]` | Edge with bigger Y of the base at -z |
| `pt[4],pt[5]` | Edge with smaller Y of the base at +z |
| `pt[6],pt[7]` | Edge with bigger Y of the base at +z |

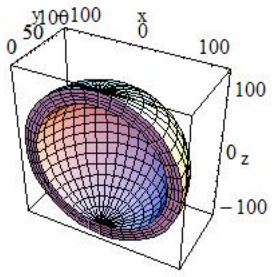
Array of vertices is given as a sequence of four edges parallel to the X-axis, first two edges define the base at -z, next two edges define the base at +z. First point in edge should have smaller X.

**Note:** The following properties of `G4Trap` should be respected: (a) Lateral faces should be planar; (b) The line joining the centers of the bases should go through the origin

**Sphere or Spherical Shell Section:**

To build a **sphere**, or a **spherical shell section**, use:

```
G4Sphere(const G4String& pName,
                G4double  pRmin,
                G4double  pRmax,
                G4double  pSPhi,
                G4double  pDPhi,
                G4double  pSTheta,
                G4double  pDTheta )
```



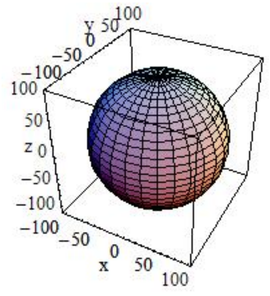
*In the picture*:
`pRmin = 100, pRmax = 120, pSPhi = 0*Degree, pDPhi = 180*Degree, pSTheta = 0 Degree, pDTheta = 180*Degree`

to obtain a solid with name `pName` and parameters:

| | |
|---|---|
| `pRmin` | Inner radius |
| `pRmax` | Outer radius |
| `pSPhi` | Starting Phi angle of the segment in radians |
| `pDPhi` | Delta Phi angle of the segment in radians |
| `pSTheta` | Starting Theta angle of the segment in radians |
| `pDTheta` | Delta Theta angle of the segment in radians |

**Full Solid Sphere:**

To build a **full solid sphere** use:

| | |
|---|---|
| <pre>G4Orb(const G4String& pName,<br>             G4double  pRmax)</pre> |  *In the picture*: `pRmax = 100` |

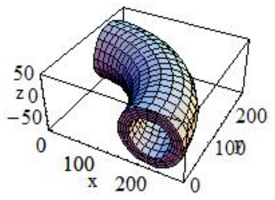
The Orb can be obtained from a Sphere with: `pRmin` = 0, `pSPhi` = 0, `pDPhi` = $2*\pi$, `pSTheta` = 0, `pDTheta` = $\pi$

| | |
|---|---|
| `pRmax` | Outer radius |

**Torus:**

To build a **torus** use:

```
G4Torus(const G4String& pName,
               G4double  pRmin,
               G4double  pRmax,
               G4double  pRtor,
               G4double  pSPhi,
               G4double  pDPhi)
```



*In the picture*:
`pRmin = 40, pRmax = 60, pRtor = 200, pSPhi = 0, pDPhi = 90*degree`

to obtain a solid with name `pName` and parameters:

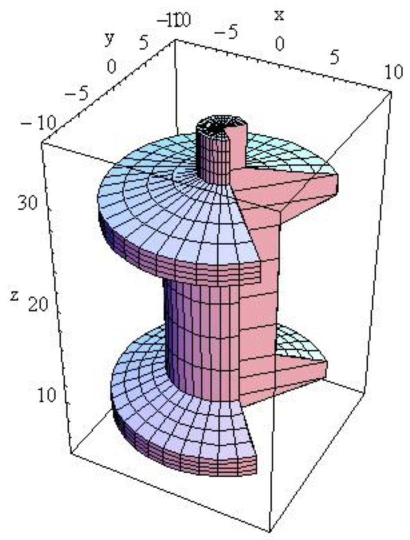
| | |
|---|---|
| `pRmin` | Inside radius |
| `pRmax` | Outside radius |
| `pRtor` | Swept radius of torus |
| `pSPhi` | Starting Phi angle in radians (`fSPhi+fDPhi<=2PI`, `fSPhi>-2PI`) |
| `pDPhi` | Delta angle of the segment in radians |

In addition, the GEANT4 Design Documentation shows in the Solids Class Diagram the complete list of CSG classes.

**Specific CSG Solids**

**Polycons:**

**Polycons** (PCON) are implemented in GEANT4 through the `G4Polycone` class:

| | |
|---|---|
| `G4Polycone(const G4String& pName, G4double phiStart, G4double phiTotal, G4int numZPlanes, const G4double zPlane[], const G4double rInner[], const G4double rOuter[])` |  *In the picture*: `phiStart = 1/4*Pi, phiTotal = 3/2*Pi, numZPlanes = 9, rInner = { 0, 0, 0, 0, 0, 0, 0, 0, 0}, rOuter = { 0, 10, 10, 5 , 5, 10 , 10 , 2, 2}, z = { 5, 7, 9, 11, 25, 27, 29, 31, 35 }` |

```
G4Polycone(const G4String& pName,
                 G4double  phiStart,
                 G4double  phiTotal,
                 G4int     numZPlanes,
           const G4double  zPlane[],
           const G4double  rInner[],
           const G4double  rOuter[])
```

where:

| | |
|---|---|
| `phiStart` | Initial Phi starting angle |
| `phiTotal` | Total Phi angle |
| `numZPlanes` | Number of Z planes |
| `numRZ` | Number of corners in r,Z space |
| `zPlane` | Position of Z planes, with Z in increasing order |
| `rInner` | Tangent distance to inner surface |
| `rOuter` | Tangent distance to outer surface |
| `r` | r coordinate of corners |
| `z` | Z coordinate of corners |

A **Polycone** where Z planes position can also decrease is implemented through the `G4GenericPolycone` class:

```
G4GenericPolycone(const G4String& pName,
                        G4double  phiStart,
                        G4double  phiTotal,
                        G4int     numRZ,
                  const G4double  r[],
                  const G4double  z[])
```

where:

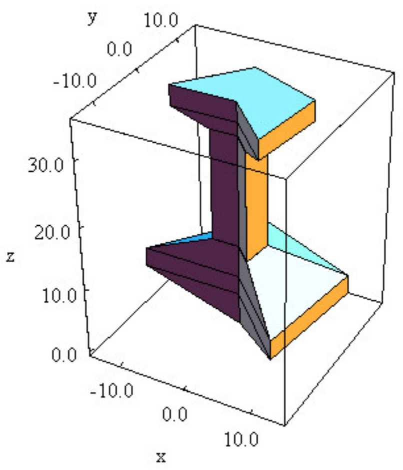
| | |
|---|---|
| `phiStart` | Initial Phi starting angle |
| `phiTotal` | Total Phi angle |
| `numRZ` | Number of corners in r,Z space |
| `r` | r coordinate of corners |
| `z` | Z coordinate of corners |

**Polyhedra (PGON):**

**Polyhedra** (PGON) are implemented through `G4Polyhedra`:

```
G4Polyhedra(const G4String& pName,
                 G4double  phiStart,
                 G4double  phiTotal,
                 G4int     numSide,
                 G4int     numZPlanes,
            const G4double  zPlane[],
            const G4double  rInner[],
            const G4double  rOuter[] )

G4Polyhedra(const G4String& pName,
                 G4double  phiStart,
                 G4double  phiTotal,
                 G4int     numSide,
                 G4int     numRZ,
            const G4double  r[],
            const G4double  z[] )
```



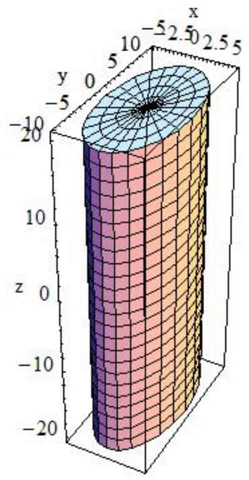
*In the picture*: `phiStart = -1/4*Pi, phiTotal= 5/4*Pi, numSide = 3, nunZPlanes = 7, rInner = { 0, 0, 0, 0, 0, 0, 0 }, rOuter = { 0, 15, 15, 4, 4, 10, 10 }, z = { 0, 5, 8, 13 , 30, 32, 35 }`

where:

| | |
|---|---|
| `phiStart` | Initial Phi starting angle |
| `phiTotal` | Total Phi angle |
| `numSide` | Number of sides |
| `numZPlanes` | Number of Z planes |
| `numRZ` | Number of corners in r,Z space |
| `zPlane` | Position of Z planes |
| `rInner` | Tangent distance to inner surface |
| `rOuter` | Tangent distance to outer surface |
| `r` | r coordinate of corners |
| `z` | Z coordinate of corners |

**Tube with an elliptical cross section:**

A **tube with an elliptical cross section** (ELTU) with elliptical semimajor and semiminor axes along the X and Y cartesian axes can be defined as follows:

| | |
|---|---|
| `G4EllipticalTube(const G4String& pName, G4double xSemiAxis, G4double ySemiAxis, G4double Dz)` |   x<br>−5<br>−2.5<br>0<br>2.5<br>5<br>y<br>10<br>5<br>0<br>−5<br>−10<br>z<br>20<br>10<br>0<br>−10<br>−20 *In the picture* `xSemiAxis = 5, semiAxisY = 10, Dz = 20` |

The tube extends in `Z` from `-Dz` to `+Dz` and the equation of the surface in the x/y plane is:

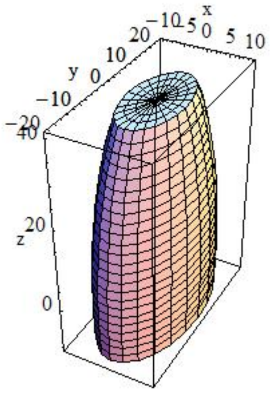
```
(x/xSemiAxis)**2+(y/ySemiAxis)**2 = 1.0
```

where:

| | |
|---|---|
| `xSemiAxis` | Half length of axis along X |
| `ySemiAxis` | Half length of axis along Y |
| `Dz` | Half length Z |

**General Ellipsoid:**

The general **ellipsoid** with possible cut in `Z` can be defined as follows:

| | |
|---|---|
| `G4Ellipsoid(const G4String& pName, G4double xSemiAxis, G4double ySemiAxis, G4double zSemiAxis, G4double zBottomCut=0, G4double zTopCut=0)` |   x<br>-10<br>-5<br>0<br>5<br>10<br>y<br>20<br>10<br>0<br>-10<br>-20<br>40<br>20<br>z<br>0 *In the picture*: `xSemiAxis = 10, ySemiAxis = 20, zSemiAxis = 50, zBottomCut = -10, pzTopCut = 40` |

```
G4Ellipsoid(const G4String& pName,
                  G4double  xSemiAxis,
                  G4double  ySemiAxis,
                  G4double  zSemiAxis,
                  G4double  zBottomCut=0,
                  G4double  zTopCut=0)
```

A general (or triaxial) ellipsoid is a quadratic surface which is given in Cartesian coordinates by:

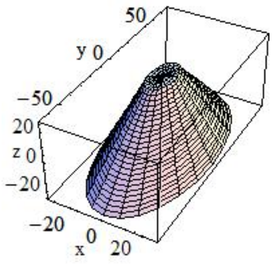
```
1.0 = (x/xSemiAxis)**2 + (y/ySemiAxis)**2 + (z/zSemiAxis)**2
```

where:

| | |
|---|---|
| `xSemiAxis` | Semiaxis in X |
| `ySemiAxis` | Semiaxis in Y |
| `zSemiAxis` | Semiaxis in Z |
| `zBottomCut` | lower cut plane level, Z |
| `zTopCut` | upper cut plane level, Z |

**Cone with Elliptical Cross Section:**

A **cone with an elliptical cross section** can be defined as follows:

| | |
|---|---|
| `G4EllipticalCone(const G4String& pName, G4double xSemiAxis, G4double ySemiAxis, G4double zHeight, G4double zTopCut)` |  *In the picture*: `xSemiAxis = 30/75, ySemiAxis = 60/75, zHeight = 50, zTopCut = 25` |

```
G4EllipticalCone(const G4String& pName,
                       G4double  xSemiAxis,
                       G4double  ySemiAxis,
                       G4double  zHeight,
                       G4double  zTopCut)
```

where:

| | |
|---|---|
| `xSemiAxis` | A scalar value, it defines the scaling along X-axis |
| `ySemiAxis` | A scalar value, it defines the scaling along Y-axis |
| `zHeight` | Z-coordinate if the apex |
| `zTopCut` | Upper cut plane level |

Value of `zTopCut` cannot exceed `zHeight`; the bases of an elliptical cone are located at `-zTopCut` and `+zTopCut`.

The lateral surface of an elliptical cone is described by the equation:

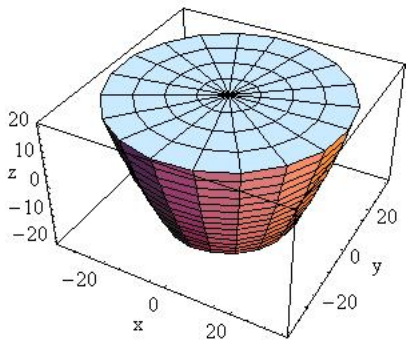
(x/xSemiAxis)**2 + (y/ySemiAxis)**2 = (zHeight - z)**2

Values of `xSemiAxis` and `ySemiAxis` can be figured out from the equations for the semimajor axes of the elliptical section at z=0:

dx = xSemiAxis * zHeight dy = ySemiAxis * zHeight

**Paraboloid, a solid with parabolic profile:**

A **solid with parabolic profile** and possible cuts along the `Z` axis can be defined as follows:

| | |
|---|---|
| `G4Paraboloid(const G4String& pName, G4double Dz, G4double R1, G4double R2)` The equation for the solid is: `rho**2 <= k1 * z + k2;` `-dz <= z <= dz` `r1**2 = k1 * (-dz) + k2` `r2**2 = k1 * ( dz) + k2` |  *In the picture*: `R1 = 20, R2 = 35, Dz = 20` |

```
G4Paraboloid(const G4String& pName,
                         G4double  Dz,
                         G4double  R1,
                         G4double  R2)
```

The equation for the solid is:

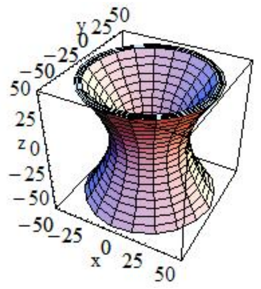
```
rho**2 <= k1 * z + k2;
    -dz <= z <= dz
r1**2 = k1 * (-dz) + k2
r2**2 = k1 * ( dz) + k2
```

| | | | | | |
|---|---|---|---|---|---|
| `Dz` | Half length Z | `R1` | Radius at -Dz | `R2` | Radius at +Dz greater than R1 |

**Tube with Hyperbolic Profile:**

A **tube with a hyperbolic profile** (HYPE) can be defined as follows:

```
G4Hype(const G4String& pName,
              G4double  innerRadius,
              G4double  outerRadius,
              G4double  innerStereo,
              G4double  outerStereo,
              G4double  halfLenZ)
```



*In the picture*:
`innerStereo = 0.7, outerStereo = 0.7, halfLenZ = 50, innerRadius = 20, outerRadius = 30`

G4Hype is shaped with curved sides parallel to the Z-axis, has a specified half-length along the Z axis about which it is centred, and a given minimum and maximum radius.

A minimum radius of 0 defines a filled Hype (with hyperbolic inner surface), i.e. inner radius = 0 AND inner stereo angle = 0.

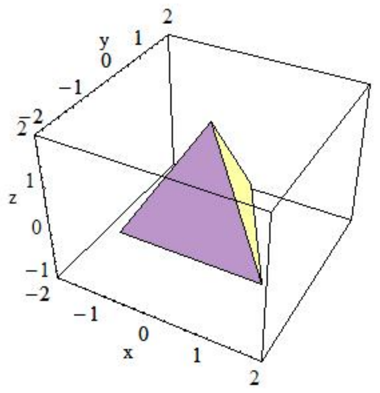
The inner and outer hyperbolic surfaces can have different stereo angles. A stereo angle of 0 gives a cylindrical surface:

| | |
|---|---|
| innerRadius | Inner radius |
| outerRadius | Outer radius |
| innerStereo | Inner stereo angle in radians |
| outerStereo | Outer stereo angle in radians |
| halfLenZ | Half length in Z |

**Tetrahedra:**

A **tetrahedra** solid can be defined as follows:

```
G4Tet(const G4String& pName,
            G4ThreeVector anchor,
            G4ThreeVector p2,
            G4ThreeVector p3,
            G4ThreeVector p4,
            G4bool* degeneracyFlag=nullptr)
```



*In the picture*: anchor = {0, 0, sqrt(3)}, p2 = { 0, 2*sqrt(2/3), -1/sqrt(3) }, p3 = { -sqrt(2), -sqrt(2/3),-1/sqrt(3) }, p4 = { sqrt(2), -sqrt(2/3) , -1/sqrt(3) }

The solid is defined by 4 points in space:

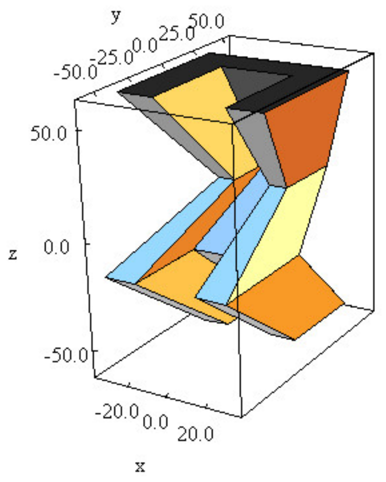
| | |
|---|---|
| anchor | Anchor point |
| p2 | Point 2 |
| p3 | Point 3 |
| p4 | Point 4 |
| degeneracyFlag | Flag indicating degeneracy of points |

**Extruded Polygon:**

The extrusion of an arbitrary polygon (**extruded solid**) with fixed outline in the defined Z sections can be defined as follows (in a general way, or in a simplified construct with only two Z sections). G4ExtrudedSolid is constructed by moving a 2D polygonal contour along a 3D polyline. During movement the polygonal contour can be scaled.

```
G4ExtrudedSolid(const G4String& pName,
        std::vector<G4TwoVector> polygon,
        std::vector<ZSection> zsections)

G4ExtrudedSolid(const G4String& pName,
     std::vector<G4TwoVector> polygon,
                      G4double halfZ,
     G4TwoVector off1, G4double scale1,
     G4TwoVector off2, G4double scale2)
```



*In the picture*:
`polygon = {-30,-30},{-30,30},{30,30},{30,-30}, {15,-30},{15,15},{-15,15},{-15,-30}`
`zsections = [-60,{0,30},0.8], [-15,{0,-30},1.], [10,{0,0},0.6], [60,{0,30},1.2]`

The Z-sides of the solid are the scaled versions of the same polygon.

| | |
|---|---|
| `polygon` | 2D polygonal contour; the vertices of the outlined polygon defined in clock-wise order |
| `zsections` | 3D polyline with scale factors; the Z-sections defined by Z position in increasing order |
| `halfZ` | Half length in Z; distance from the origin to the sections |
| `off1, scale1` | (X, Y) position of the polygon and scale factor at -halfZ |
| `off2, scale2` | (X, Y) position of the polygon and scale factor at +halfZ |

Each node in the 3D polyline is defined as a ZSection object:

```
struct ZSection
{
  G4double fZ;          // Z coordinate of the node
  G4TwoVector fOffset; // (X, Y) coordinates of the node
  G4double fScale;      // Scale factor that should be applied to the 2D polygon at the node
}
```

Very often an **extruded solid** is constructed by shifting a polygon in the perpendicular direction to its plane. In such case `off1`, `off2` should be specified as G4TwoVector(0,0) and `scale1`, `scale2` should be equal to 1.

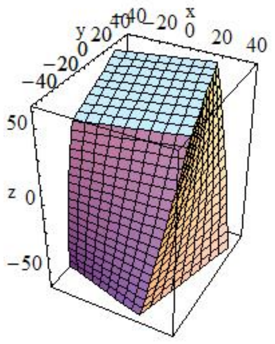
**Box Twisted:**

A **box twisted** along one axis can be defined as follows:

<table>
<tr><td>

```
G4TwistedBox(const G4String& pName,
                   G4double  twistedangle,
                   G4double  pDx,
                   G4double  pDy,
                   G4double  pDz)
```

</td><td>



*In the picture*:
`twistedangle = 30*Degree, pDx = 30, pDy =40, pDz = 60`

</td></tr>
</table>

`G4TwistedBox` is a box twisted along the z-axis. The twist angle cannot be greater than 90 degrees:

| | |
|---|---|
| `twistedangle` | Twist angle |
| `pDx` | Half x length |
| `pDy` | Half y length |
| `pDz` | Half z length |

**Trapezoid Twisted along One Axis:**

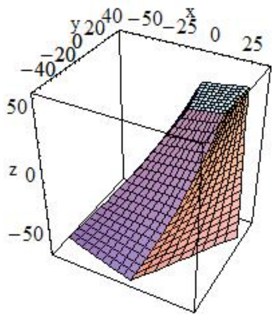
*trapezoid twisted* along one axis can be defined as follows:

<table>
<tr><td>

```
G4TwistedTrap(const G4String& pName,
                    G4double  twistedangle,
                    G4double  pDxx1,
                    G4double  pDxx2,
                    G4double  pDy,
                    G4double  pDz)

G4TwistedTrap(const G4String& pName,
                    G4double  twistedangle,
                    G4double  pDz,
                    G4double  pTheta,
                    G4double  pPhi,
                    G4double  pDy1,
                    G4double  pDx1,
                    G4double  pDx2,
                    G4double  pDy2,
                    G4double  pDx3,
                    G4double  pDx4,
                    G4double  pAlph)
```

</td><td>



*In the picture*:
`pDx1 = 30, pDx2 = 40, pDy1 = 40, pDx3 = 10, pDx4 = 14, pDy2 = 16, pDz = 60, pTheta = 20*Degree, pDphi = 5*Degree, pAlph = 10*Degree, twistedangle = 30*Degree`

</td></tr>
</table>

The first constructor of `G4TwistedTrap` produces a regular trapezoid twisted along the `Z`-axis, where the caps of the trapezoid are of the same shape and size.

The second constructor produces a generic trapezoid with polar, azimuthal and tilt angles.

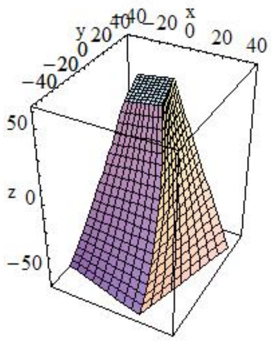
The twist angle cannot be greater than 90 degrees:

| | |
|---|---|
| `twistedangle` | Twisted angle |
| `pDx1` | Half X length at y=-pDy |
| `pDx2` | Half X length at y=+pDy |
| `pDy` | Half Y length |
| `pDz` | Half Z length |
| `pTheta` | Polar angle of the line joining the centres of the faces at -/+pDz |
| `pDy1` | Half Y length at -pDz |
| `pDx1` | Half X length at -pDz, y=-pDy1 |
| `pDx2` | Half X length at -pDz, y=+pDy1 |
| `pDy2` | Half Y length at +pDz |
| `pDx3` | Half X length at +pDz, y=-pDy2 |
| `pDx4` | Half X length at +pDz, y=+pDy2 |
| `pAlph` | Angle with respect to the Y axis from the centre of the side |

**Twisted Trapezoid with X and Y dimensions varying along Z:**

A **twisted trapezoid** with the `X` and `Y` dimensions **varying along** `Z` can be defined as follows:

```
G4TwistedTrd(const G4String& pName,
                   G4double  pDx1,
                   G4double  pDx2,
                   G4double  pDy1,
                   G4double  pDy2,
                   G4double  pDz,
                   G4double  twistedangle)
```



*In the picture*:
`dx1 = 30, dx2 = 10,   dy1 = 40, dy2 = 15, dz = 60, twistedangle = 30*Degree`

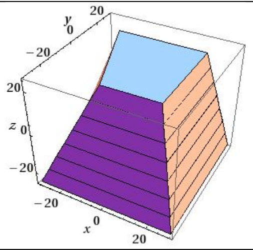
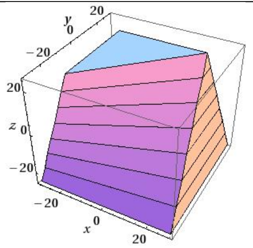
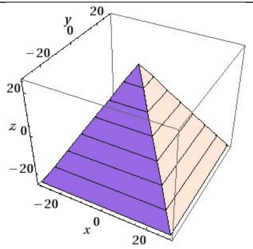
where:

| | |
|---|---|
| `pDx1` | Half X length at the surface positioned at -dz |
| `pDx2` | Half X length at the surface positioned at +dz |
| `pDy1` | Half Y length at the surface positioned at -dz |
| `pDy2` | Half Y length at the surface positioned at +dz |
| `pDz` | Half Z length |
| `twistedangle` | Twisted angle |

**Generic trapezoid with optionally collapsing vertices:**

An **arbitrary trapezoid** with up to 8 vertices standing on two parallel planes perpendicular to the `Z` axis can be defined as follows:

```
G4GenericTrap(const G4String& pName,
                    G4double  pDz,
              const std::vector<G4TwoVector>& vertices)
```

| |
|---|
|  |
| *In the picture*: `pDz = 25   vertices = {-30, -30}, {-30, 30}, {30, 30}, {30, -30}   {-5, -20}, {-20, 20}, {20, 20}, {20, -20}` |
| |
|  |
| *In the picture*: `pDz = 25  vertices = {-30,-30}, {-30,30}, {30,30}, {30,-30}  {-20,-20},{-20, 20}, {20,20}, {20, 20}` |
| |
|  |
| *In the picture*: `pDz = 25 vertices = {-30,-30}, {-30,30}, {30,30}, {30,-30} {0,0}, {0,0}, {0, 0}, {0,0}` |

where:

| | |
|---|---|
| `pDz` | Half Z length |
| `vertices` | The (X,Y) coordinates of vertices |

The order of specification of the coordinates for the vertices in `G4GenericTrap` is important. The first four points are the vertices sitting on the `-hz` plane; the last four points are the vertices sitting on the `+hz` plane.

The order of defining the vertices of the solid is the following:

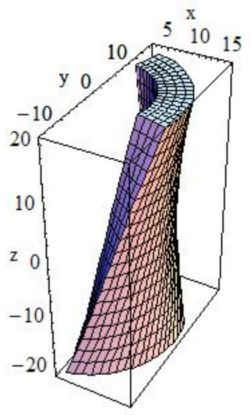
```
point 0 is connected with points 1,3,4
point 1 is connected with points 0,2,5
point 2 is connected with points 1,3,6
point 3 is connected with points 0,2,7
point 4 is connected with points 0,5,7
point 5 is connected with points 1,4,6
point 6 is connected with points 2,5,7
point 7 is connected with points 3,4,6
```

Points can be identical in order to create shapes with less than 8 vertices; the only limitation is to have at least one triangle at `+hz` or `-hz`; the lateral surfaces are not necessarily planar. Not planar lateral surfaces are represented by a surface that linearly changes from the edge on `-hz` to the corresponding edge on `+hz`; it represents a *sweeping* surface with twist angle linearly dependent on `Z`, but it is not a real twisted surface mathematically described by equations as for the other *twisted* solids described in this chapter.

**Tube Section Twisted along Its Axis:**

A **tube section twisted** along its axis can be defined as follows:

```
G4TwistedTubs(const G4String& pName,
                     G4double  twistedangle,
                     G4double  endinnerrad,
                     G4double  endouterrad,
                     G4double  halfzlen,
                     G4double  dphi)
```



*In the picture*:
`endinnerrad = 10, endouterrad = 15, halfzlen = 20, dphi = 90*Degree, twistedangle = 60*Degree`

`G4TwistedTubs` is a sort of twisted cylinder which, placed along the `Z`-axis and divided into `phi`-segments is shaped like an hyperboloid, where each of its segmented pieces can be tilted with a stereo angle.

It can have inner and outer surfaces with the same stereo angle:

| | |
|---|---|
| `twistedangle` | Twisted angle |
| `endinnerrad` | Inner radius at endcap |
| `endouterrad` | Outer radius at endcap |
| `halfzlen` | Half Z length |
| `dphi` | Phi angle of a segment |

Additional constructors are provided, allowing the shape to be specified either as:

- the number of segments in `phi` and the total angle for all segments, or
- a combination of the above constructors providing instead the inner and outer radii at `z=0` with different `Z`-lengths along negative and positive `Z`-axis.

### Solids made by Boolean operations

Simple solids can be combined using Boolean operations. For example, a cylinder and a half-sphere can be combined with the union Boolean operation.

Creating such a new *Boolean* solid, requires:

- Two solids
- A Boolean operation: union, intersection or subtraction.
- Optionally a transformation for the second solid.

The solids used should be either CSG solids (for examples a box, a spherical shell, or a tube) or another Boolean solid: the product of a previous Boolean operation. An important purpose of Boolean solids is to allow the description of solids with peculiar shapes in a simple and intuitive way, still allowing an efficient geometrical navigation inside them.

**Note:** The constituent solids of a Boolean operation should possibly *avoid* be composed by sharing all or part of their surfaces. This precaution is necessary in order to avoid the generation of 'fake' surfaces due to precision loss, or errors in the final visualization of the Boolean shape. In particular, if any one of the *subtractor* surfaces is coincident with a surface of the *subtractee*, the result is undefined. Moreover, the final Boolean solid should represent a single 'closed' solid, i.e. a Boolean operation between two solids which are disjoint or far apart each other, is *not* a valid Boolean composition.

**Note:** The tracking cost for navigating in a Boolean solid is proportional to the number of constituent solids. So care must be taken to avoid extensive, unnecessary use of Boolean solids in performance-critical areas of a geometry description, where each solid is created from Boolean combinations of many other solids.

Examples of the creation of the simplest Boolean solids are given below:

```
G4Box*  box =
  new G4Box("Box",20*mm,30*mm,40*mm);
G4Tubs* cyl =
  new G4Tubs("Cylinder",0,50*mm,50*mm,0,twopi);  // r:      0 mm -> 50 mm
                                                 // z:    -50 mm -> 50 mm
                                                 // phi:    0 ->   2 pi
G4UnionSolid* union =
  new G4UnionSolid("Box+Cylinder", box, cyl);
G4IntersectionSolid* intersection =
  new G4IntersectionSolid("Box*Cylinder", box, cyl);
G4SubtractionSolid* subtraction =
  new G4SubtractionSolid("Box-Cylinder", box, cyl);
```

where the union, intersection and subtraction of a box and cylinder are constructed.

The more useful case where one of the solids is displaced from the origin of coordinates also exists. In this case the second solid is positioned relative to the coordinate system (and thus relative to the first). This can be done in two ways:

- Either by giving a rotation matrix and translation vector that are used to transform the coordinate system of the second solid to the coordinate system of the first solid. This is called the *passive* method.
- Or by creating a transformation that moves the second solid from its desired position to its standard position, e.g., a box's standard position is with its centre at the origin and sides parallel to the three axes. This is called the *active* method.

In the first case, the translation is applied first to move the origin of coordinates. Then the rotation is used to rotate the coordinate system of the second solid to the coordinate system of the first.

```
G4RotationMatrix* yRot = new G4RotationMatrix;  // Rotates X and Z axes only
yRot->rotateY(M_PI/4.*rad);                     // Rotates 45 degrees
G4ThreeVector zTrans(0, 0, 50);

G4UnionSolid* unionMoved =
  new G4UnionSolid("Box+CylinderMoved", box, cyl, yRot, zTrans);
//
// The new coordinate system of the cylinder is translated so that
// its centre is at +50 on the original Z axis, and it is rotated
// with its X axis halfway between the original X and Z axes.

// Now we build the same solid using the alternative method
//
G4RotationMatrix invRot = yRot->invert();
G4Transform3D transform(invRot, zTrans);
G4UnionSolid* unionMoved =
  new G4UnionSolid("Box+CylinderMoved", box, cyl, transform);
```

Note that the first constructor that takes a pointer to the rotation-matrix (`G4RotationMatrix*`), does NOT copy it. Therefore once used a rotation-matrix to construct a Boolean solid, it must NOT be modified.

In contrast, with the alternative method shown, a `G4Transform3D` is provided to the constructor by value, and its transformation is stored by the Boolean solid. The user may modify the `G4Transform3D` and eventually use it again.

When positioning a volume associated to a Boolean solid, the relative center of coordinates considered for the positioning is the one related to the *first* of the two constituent solids.

### Multi-Union Structures

Since release 10.4, the possibility to define multi-union structures is part of the standard set of constructs in GEANT4. A `G4MultiUnion` structure allows for the description of a Boolean union of many displaced solids at once, therefore representing volumes with the same associated material. An example on how to define a simple MultiUnion structure is given here:

```
#include "G4MultiUnion.hh"

// Define two -G4Box- shapes
//
G4Box* box1 = new G4Box("Box1", 5.*mm, 5.*mm, 10.*mm);
G4Box* box2 = new G4Box("Box2", 5.*mm, 5.*mm, 10.*mm);

// Define displacements for the shapes
//
G4RotationMatrix rotm  = G4RotationMatrix();
G4ThreeVector position1 = G4ThreeVector(0.,0.,1.);
G4ThreeVector position2 = G4ThreeVector(0.,0.,2.);
G4Transform3D tr1 = G4Transform3D(rotm,position1);
G4Transform3D tr2 = G4Transform3D(rotm,position2);

// Initialise a MultiUnion structure
//
G4MultiUnion* munion_solid = new G4MultiUnion("Boxes_Union");

// Add the shapes to the structure
//
munion_solid->AddNode(*box1,tr1);
munion_solid->AddNode(*box2,tr2);

// Finally close the structure
//
munion_solid->Voxelize();
```

(continues on next page)

(continued from previous page)

```
// Associate it to a logical volume as a normal solid
//
G4LogicalVolume* lvol =
new G4LogicalVolume(munion_solid,          // its solid
                    munion_mat,            // its material
                    "Boxes_Union_LV");     // its name
```

Fast detection of intersections in tracking is assured by the adoption of a specialised optimisation applied to the 3D structure itself and generated at initialisation.

## Tessellated Solids

In GEANT4 it is also implemented a class `G4TessellatedSolid` which can be used to generate a generic solid defined by a number of facets (`G4VFacet`). Such constructs are especially important for conversion of complex geometrical shapes imported from CAD systems bounded with generic surfaces into an approximate description with facets of defined dimension (see Fig. 4.1).

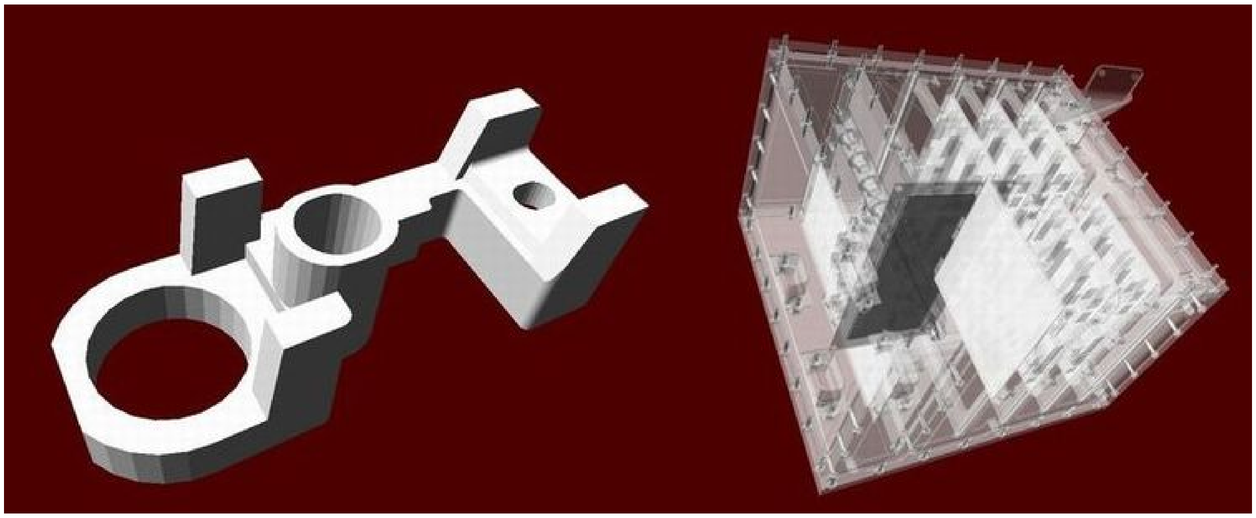


Fig. 4.1: Example of geometries imported from CAD system and converted to tessellated solids.

They can also be used to generate a solid bounded with a generic surface made of planar facets. It is important that the supplied facets shall form a fully enclosed space to represent the solid, and that adjacent facets always share a complete edge (no vertex on one facet can lie between vertices on an adjacent facet).

Two types of facet can be used for the construction of a `G4TessellatedSolid`: a triangular facet (`G4TriangularFacet`) and a quadrangular facet (`G4QuadrangularFacet`).

An example on how to generate a simple tessellated shape is given below.

Listing 4.1: Example of geometries imported from CAD system and converted to tessellated solids.

```
// First declare a tessellated solid
//
G4TessellatedSolid solidTarget = new G4TessellatedSolid("Solid_name");

// Define the facets which form the solid
//
G4double targetSize = 10*cm ;
```

(continues on next page)

(continued from previous page)

```
G4TriangularFacet *facet1 = new
G4TriangularFacet (G4ThreeVector(-targetSize,-targetSize,        0.0),
                   G4ThreeVector(+targetSize,-targetSize,        0.0),
                   G4ThreeVector(        0.0,        0.0,+targetSize),
                   ABSOLUTE);
G4TriangularFacet *facet2 = new
G4TriangularFacet (G4ThreeVector(+targetSize,-targetSize,        0.0),
                   G4ThreeVector(+targetSize,+targetSize,        0.0),
                   G4ThreeVector(        0.0,        0.0,+targetSize),
                   ABSOLUTE);
G4TriangularFacet *facet3 = new
G4TriangularFacet (G4ThreeVector(+targetSize,+targetSize,        0.0),
                   G4ThreeVector(-targetSize,+targetSize,        0.0),
                   G4ThreeVector(        0.0,        0.0,+targetSize),
                   ABSOLUTE);
G4TriangularFacet *facet4 = new
G4TriangularFacet (G4ThreeVector(-targetSize,+targetSize,        0.0),
                   G4ThreeVector(-targetSize,-targetSize,        0.0),
                   G4ThreeVector(        0.0,        0.0,+targetSize),
                   ABSOLUTE);
G4QuadrangularFacet *facet5 = new
G4QuadrangularFacet (G4ThreeVector(-targetSize,-targetSize,        0.0),
                     G4ThreeVector(-targetSize,+targetSize,        0.0),
                     G4ThreeVector(+targetSize,+targetSize,        0.0),
                     G4ThreeVector(+targetSize,-targetSize,        0.0),
                     ABSOLUTE);

// Now add the facets to the solid
//
solidTarget->AddFacet((G4VFacet*) facet1);
solidTarget->AddFacet((G4VFacet*) facet2);
solidTarget->AddFacet((G4VFacet*) facet3);
solidTarget->AddFacet((G4VFacet*) facet4);
solidTarget->AddFacet((G4VFacet*) facet5);

Finally declare the solid is complete
//
solidTarget->SetSolidClosed(true);
```

The `G4TriangularFacet` class is used for the construction of `G4TessellatedSolid`. It is defined by three vertices, which shall be supplied in *anti-clockwise order* looking from the outside of the solid where it belongs. Its constructor looks like:

```
G4TriangularFacet ( const G4ThreeVector     Pt0,
                    const G4ThreeVector     vt1,
                    const G4ThreeVector     vt2,
                          G4FacetVertexType fType )
```

i.e., it takes 4 parameters to define the three vertices:

| | |
|---|---|
| `G4FacetVertexType` | `ABSOLUTE` in which case `Pt0`, `vt1` and `vt2` are the three vertices in anti-clockwise order looking from the outside. |
| `G4FacetVertexType` | `RELATIVE` in which case the first vertex is `Pt0`, the second vertex is `Pt0+vt1` and the third vertex is `Pt0+vt2`, all in anti-clockwise order when looking from the outside. |

The `G4QuadrangularFacet` class can be used for the construction of `G4TessellatedSolid` as well. It is defined by four vertices, which shall be in the same plane and be supplied in *anti-clockwise order* looking from the outside of the solid where it belongs. Its constructor looks like:

```
G4QuadrangularFacet ( const G4ThreeVector     Pt0,
                      const G4ThreeVector     vt1,
                      const G4ThreeVector     vt2,
                      const G4ThreeVector     vt3,
                            G4FacetVertexType fType )
```

i.e., it takes 5 parameters to define the four vertices:

| | |
|---|---|
| `G4FacetVertexType` | `ABSOLUTE` in which case `Pt0`, `vt1`, `vt2` and `vt3` are the four vertices required in anti-clockwise order when looking from the outside. |
| `G4FacetVertexType` | `RELATIVE` in which case the first vertex is `Pt0`, the second vertex is `Pt0+vt`, the third vertex is `Pt0+vt2` and the fourth vertex is `Pt0+vt3`, in anti-clockwise order when looking from the outside. |

### Importing CAD models as tessellated shapes

Tessellated solids can also be used to import geometrical models from CAD systems (see *fig-geom-solid-1*). In order to do this, it is required to convert first the CAD shapes into tessellated surfaces. A way to do this is to save the shapes in the geometrical model as STEP files and convert them to tessellated (faceted surfaces) solids, using a tool which allows such conversion. This strategy allows to import any shape with some degree of approximation; the converted CAD models can then be imported through GDML (Geometry Description Markup Language) into GEANT4 and be represented as `G4TessellatedSolid` shapes.

Tools which can be used to generate meshes to be then imported in GEANT4 as tessellated solids are:

- FASTRAD - 3D tool for radiation shielding analysis; exports meshes to GDML.
- InStep - A free STL to GDML conversion tool.
- SALOME - Open-source software allowing to import STEP/BREP/IGES/STEP/ACIS formats, mesh them and export to STL.
- ESABASE2 - Space environment analysis CAD, basic modules free for academic non-commercial use. Can import STEP files and export to GDML shapes or complete geometries.
- CADMesh - Tool based on the VCG Library to read STL files and import in GEANT4.
- Cogenda - Commercial TCAD software for generation of 3D meshes through the module *Gds2Mesh* and final export to GDML.
- EDGE - A commercial GDML editor, able to import/export STEP/STL geometries.
- CadMC - Tool to convert FreeCAD geometries to Geant4 (tessellated and CSG shapes).
- pyg4ometry - A python library to manipulate GDML geometery. Has an interface from OpenCASCADE and Geant4 tessellated

### Unified Solids

An alternative implementation for most of the cited geometrical primitives is provided since release 10.0 of GEANT4. With release 10.6, all primitives shapes except the twisted specific solids, can be replaced.

The code for the new geometrical primitives originated as part of the AIDA Unified Solids Library and is now integrated in the VecGeom library (the vectorized geometry library for particle-detector simulation); it is provided as alternative use and can be activated in place of the original primitives defined in GEANT4, by selecting the appropriate compilation flag when configuring the GEANT4 libraries installation. The installation allows to build against an external system installation of the VecGeom library, therefore the appropriate installation path must also be provided during the installation configuration:

```
-DGEANT4_USE_USOLIDS="all"      // to replace all available shapes
-DGEANT4_USE_USOLIDS="box;tubs" // to replace only individual shapes
```

The original API for all geometrical primitives is preserved.

### 4.1.3 Logical Volumes

The Logical Volume manages the information associated with detector elements represented by a given Solid and Material, independently from its physical position in the detector.

`G4LogicalVolumes` must be allocated using 'new' in the user's program; they get registered to a `G4LogicalVolumeStore` at construction, which will also take care to deallocate them at the end of the job, if not done already in the user's code.

A Logical Volume knows which physical volumes are contained within it. It is uniquely defined to be their mother volume. A Logical Volume thus represents a hierarchy of unpositioned volumes whose positions relative to one another are well defined. By creating Physical Volumes, which are placed instances of a Logical Volume, this hierarchy or tree can be repeated.

A Logical Volume also manages the information relative to the Visualization attributes (*Visualization Attributes*) and user-defined parameters related to tracking, electro-magnetic field or cuts (through the `G4UserLimits` interface).

By default, tracking optimization of the geometry (voxelization) is applied to the volume hierarchy identified by a logical volume. It is possible to change the default behavior by choosing not to apply geometry optimization for a given logical volume. This feature does not apply to the case where the associated physical volume is a parameterised volume; in this case, optimization is always applied.

```
G4LogicalVolume( G4VSolid*              pSolid,
                 G4Material*            pMaterial,
                 const G4String&        Name,
                 G4FieldManager*        pFieldMgr=0,
                 G4VSensitiveDetector* pSDetector=0,
                 G4UserLimits*          pULimits=0,
                 G4bool                 Optimise=true )
```

**Note:** GEANT4 does not impose any restriction on the name assigned to logical volumes; names can be shared. It is however good practice to specify unique names for each logical volume, to allow for easier retrivial from stores for post-processing use.

Through the logical volume it is also possible to *tune* the granularity of the optimisation algorithm to be applied to the sub-tree of volumes represented. This is possible using the methods:

```
G4double GetSmartless() const
void SetSmartless(G4double s)
```

The default *smartless* value is *2* and controls the average number of slices per contained volume which are used in the optimisation. The smaller the value, the less fine grained optimisation grid is generated; this will translate in a possible reduction of memory consumed for the optimisation of that portion of geometry at the price of a slight CPU time increase at tracking time. Manual tuning of the optimisation is in general not required, since the optimal granularity level is computed automatically and adapted to the specific geometry setup; however, in some cases (like geometry portions with 'dense' concentration of volumes distributed in a non-uniform way), it may be necessary to adopt manual tuning for helping the optimisation process in dealing with the most critical areas. By setting the verbosity to *2* through the following UI run-time command:

```
/run/verbose 2
```

a statistics of the memory consumed for the allocated optimisation nodes will be displayed volume by volume, allowing to easily identify the critical areas which may eventually require manual intervention.

The logical volume provides a way to estimate the *mass* of a tree of volumes defining a detector or sub-detector. This can be achieved by calling the method:

```
G4double GetMass(G4bool forced=false)
```

The mass of the logical volume tree is computed from the estimated geometrical volume of each solid and material associated with the logical volume and its daughters. Note that this computation may require a considerable amount of time, depending on the complexity of the geometry tree. The returned value is cached by default and can be used for successive calls, unless recomputation is forced by providing `true` for the Boolean argument `forced` in input. Computation should be forced if the geometry setup has changed after the previous call.

Finally, the Logical Volume manages the information relative to the Envelopes hierarchy required for fast Monte Carlo parameterisations (*Parameterisation*).

### Sub-detector Regions

In complex geometry setups, such as those found in large detectors in particle physics experiments, it is useful to think of specific Logical Volumes as representing parts (sub-detectors) of the entire detector setup which perform specific functions. In such setups, the processing speed of a real simulation can be increased by assigning specific production *cuts* to each of these detector parts. This allows a more detailed simulation to occur only in those regions where it is required.

The concept of detector *Region* is introduced to address this need. Once the final geometry setup of the detector has been defined, a region can be specified by constructing it with:

```
G4Region( const G4String& rName )
```

where:

| | |
|---|---|
| `rName` | String identifier for the detector region |

`G4Regions` must be allocated using 'new' in the user's program; they get registered to a `G4RegionStore` at construction, which will also take care to deallocate them at the end of the job, if not done already in the user's code.

A `G4Region` must then be assigned to a logical volume, in order to make it a *Root Logical Volume*:

```
G4Region* emCalorimeter = new G4Region("EM-Calorimeter");
emCalorimeterLV->SetRegion(emCalorimeter);
emCalorimeter->AddRootLogicalVolume(emCalorimeterLV);
```

A root logical volume is the first volume at the top of the hierarchy to which a given region is assigned. Once the region is assigned to the root logical volume, the information is automatically propagated to the volume tree, so that each daughter volume shares the same region. Propagation on a tree branch will be interrupted if an already existing root logical volume is encountered.

**Note:** It is recommended to assign unique names to logical volumes specified as root logical volumes, as this will guarantee proper retrievial from the store for post-processing use in persistency. The same applies for names assigned to regions.

A specific *Production Cut* can be assigned to the region, by defining and assigning to it a `G4ProductionCut` object

```
emCalorimeter->SetProductionCuts(emCalCuts);
```

*Set production threshold (SetCut methods)* describes how to define a production cut. The same region can be assigned to more than one root logical volume, and root logical volumes can be removed from an existing region. A logical volume can have only *one* region assigned to it. Regions will be automatically registered in a store which will take care of destroying them at the end of the job. A default region with a default production cut is automatically created and assigned to the world volume.

Regions can also become 'envelopes' for fast-simulation; can be assigned user-limits or generic user-information (`G4VUserRegionInformation`); can be associated to specific stepping-actions (`G4UserSteppingAction`) or have assigned a local magnetic-field (local fields specifically associated to logical volumes take precedence anyhow).

### 4.1.4 Physical Volumes

Physical volumes represent the spatial positioning of the volumes describing the detector elements. Several techniques can be used. They range from the simple placement of a single copy to the repeated positioning using either a simple linear formula or a user specified function.

Any physical volume must be allocated using 'new' in the user's program; they get registered to a `G4PhysicalVolumeStore` at construction, which will also take care to deallocate them at the end of the job, if not done already in the user's code.

The simple placement involves the definition of a transformation matrix for the volume to be positioned. Repeated positioning is defined using the number of times a volume should be replicated at a given distance along a given direction. Finally it is possible to define a parameterised formula to specify the position of multiple copies of a volume. Details about these methods are given below.

**Note:** For geometries which vary between runs and for which components of the old geometry setup are explicitly -deleted-, it is required to consider the proper order of deletion (which is the exact inverse of the actual construction, i.e., first delete physical volumes and then logical volumes). Deleting a logical volume does NOT delete its daughter volumes.

It is not necessary to delete the geometry setup at the end of a job, the system will take care to free the volume and solid stores at the end of the job. The user has to take care of the deletion of any additional transformation or rotation matrices allocated dynamically in his/her own application.

**Note:** GEANT4 does not impose any restriction on the name assigned to volumes; names can be shared. It is however good practice to specify unique names for each physical node in a tree, to allow for easier retrivial from stores for post-processing use.

#### Placements: single positioned copy

In this case, the Physical Volume is created by associating a Logical Volume with a Transformation that defines the position of the current volume in the mother volume. The solid itself is moved by rotating and translating it to bring it into the system of coordinates of the mother volume. The decomposition of the Transformation must contain only rotation and translation (reflection and scaling are not allowed).

To create a Placement one must construct it using:

```
G4PVPlacement(       G4Transform3D      solidTransform,
                     G4LogicalVolume*   pCurrentLogical,
               const G4String&          pName,
                     G4LogicalVolume*   pMotherLogical,
                     G4bool             pMany,
                     G4int              pCopyNo,
                     G4bool             pSurfChk=false )
```

where:

| | |
|---|---|
| `solidTransform` | Position in its mother volume |
| `pCurrentLogical` | The associated Logical Volume |
| `pName` | String identifier for this placement |
| `pMotherLogical` | The associated mother volume |
| `pMany` | For future use. Can be set to false |
| `pCopyNo` | Integer which identifies this placement |
| `pSurfChk` | if true activates check for overlaps with existing volumes |

Currently Boolean operations are not implemented at the level of physical volume. So `pMany` must be false. However, an alternative implementation of Boolean operations exists. In this approach a solid can be created from the union, intersection or subtraction of two solids. See *Solids made by Boolean operations* above for an explanation of this.

The mother volume must be specified for all volumes *except* the world volume.

An alternative way to specify a Placement is to use a Rotation Matrix and a Translation Vector. If compared with the previous construct, the Rotation Matrix is the inverse of the rotation from the decomposition of the transformation, but the Translation Vector is the same. The Rotation Matrix represents the rotation of the reference frame of the considered volume relatively to its mother volume's reference frame. The Translation Vector represents the translation of the current volume in the reference frame of its mother volume. This *passive* method can be utilized using the following constructor:

```
G4PVPlacement(       G4RotationMatrix*  pRot,
               const G4ThreeVector&     tlate,
                     G4LogicalVolume*   pCurrentLogical,
               const G4String&          pName,
                     G4LogicalVolume*   pMotherLogical,
                     G4bool             pMany,
                     G4int              pCopyNo,
                     G4bool             pSurfChk=false )
```

where:

| | |
|---|---|
| `pRot` | Rotation with respect to its mother volume |
| `tlate` | Translation with respect to its mother volume |
| `pCurrentLogical` | The associated Logical Volume |
| `pName` | String identifier for this placement |
| `pMotherLogical` | The associated mother volume |
| `pMany` | For future use. Can be set to false |
| `pCopyNo` | Integer which identifies this placement |
| `pSurfChk` | if true activates check for overlaps with existing volumes |

Care must be taken because the rotation matrix is not copied by a `G4PVPlacement`. So the user must not modify it after creating a Placement that uses it. However the same rotation matrix can be re-used for many volumes.

An alternative method to specify the mother volume is to specify its placed physical volume. It can be used in either of the above methods of specifying the placement's position and rotation. The effect will be exactly the same as for using the mother logical volume.

Note that a Placement Volume can still represent multiple detector elements. This can happen if several copies exist of the mother logical volume. Then different detector elements will belong to different branches of the tree of the hierarchy of geometrical volumes.

An example demonstrating various ways of placement and constructing the rotation matrix is provided in `examples/extended/geometry/transforms`.

### Repeated volumes

In this case, a single Physical Volume represents multiple copies of a volume within its mother volume, allowing to save memory. This is normally done when the volumes to be positioned follow a well defined rotational or translational symmetry along a Cartesian or cylindrical coordinate. The Repeated Volumes technique is available for most volumes described by CSG solids.

### Replicas

Replicas are *repeated volumes* in the case when the multiple copies of the volume are all identical. The coordinate axis and the number of replicas need to be specified for the program to compute at run time the transformation matrix corresponding to each copy.

```
G4PVReplica( const G4String&          pName,
                   G4LogicalVolume*   pCurrentLogical,
                   G4LogicalVolume*   pMotherLogical, // OR G4VPhysicalVolume*
             const EAxis              pAxis,
             const G4int              nReplicas,
             const G4double           width,
             const G4double           offset=0 )
```

where:

| | |
|---|---|
| `pName` | String identifier for the replicated volume |
| `pCurrentLogical` | The associated Logical Volume |
| `pMotherLogical` | The associated mother volume |
| `pAxis` | The axis along with the replication is applied |
| `nReplicas` | The number of replicated volumes |
| `width` | The width of a single replica along the axis of replication |
| `offset` | Possible offset associated to mother offset along the axis of replication |

`G4PVReplica` represents `nReplicas` volumes differing only in their positioning, and completely **filling** the containing mother volume. Consequently if a `G4PVReplica` is 'positioned' inside a given mother it **MUST** be the mother's only daughter volume. Replica's correspond to divisions or slices that completely fill the mother volume and have no offsets. For Cartesian axes, slices are considered perpendicular to the axis of replication.

The replica's positions are calculated by means of a linear formula. Replication may occur along:

- *Cartesian axes* `(kXAxis,kYAxis,kZAxis)`
  The replications, of specified width have coordinates of form `(-width*(nReplicas-1)*0.5+n*width,0,0)` where `n=0.. nReplicas-1` for the case of `kXAxis`, and are unrotated.
- *Radial axis (cylindrical polar)* `(kRho)`
  The replications are cons/tubs sections, centred on the origin and are unrotated.
  They have radii of `width*n+offset` to `width*(n+1)+offset` where `n=0..nReplicas-1`
- *Phi axis (cylindrical polar)* `(kPhi)`
  The replications are *phi sections* or *wedges*, and of cons/tubs form.
  They have `phi` of `offset+n*width` to `offset+(n+1)*width` where `n=0..nReplicas-1`

The coordinate system of the replicas is at the centre of each replica for the Cartesian axis. For the radial case, the coordinate system is unchanged from the mother. For the `phi` axis, the new coordinate system is rotated such that the X axis bisects the angle made by each wedge, and Z remains parallel to the mother's Z axis.

The solid associated via the replicas' logical volume should have the dimensions of the first volume created and must be of the correct symmetry/type, in order to assist in good visualisation.

ex. For X axis replicas in a box, the solid should be another box with the dimensions of the replications. (same Y & Z dimensions as mother box, X dimension = mother's X dimension/nReplicas).

Replicas may be placed inside other replicas, provided the above rule is observed. Normal placement volumes may be placed inside replicas, provided that they do not intersect the mother's or any previous replica's boundaries. Parameterised volumes may not be placed inside.

Because of these rules, it is not possible to place any other volume inside a replication in `radius`.

The world volume *cannot* act as a replica, therefore it cannot be sliced.

During tracking, the translation + rotation associated with each `G4PVReplica` object is modified according to the currently 'active' replication. The solid is not modified and consequently has the wrong parameters for the cases of `phi` and `r` replication and for when the cross-section of the mother is not constant along the replication.

Example

Listing 4.2: An example of simple replicated volumes with `G4PVReplica`.

```
G4PVReplica repX("Linear Array",
                 pRepLogical,
                 pContainingMotherBox,
                 kXAxis, 5, 10*mm);

G4PVReplica repR("RSlices",
                 pRepRLogical,
                 pContainingMotherTub,
                 kRho, 5, 10*mm, 0);

G4PVReplica repZ("ZSlices",
                 pRepZLogical,
                 pContainingMotherTub,
                 kZAxis, 5, 10*mm);

G4PVReplica repPhi("PhiSlices",
                   pRepPhiLogical,
                   pContainingMotherTub,
                   kPhi, 4, M_PI*0.5*rad, 0);
```

`RepX` is an array of 5 replicas of width 10*mm, positioned inside and completely filling the volume pointed by `pContainingMotherBox`. The mother's X length must be 5*10*mm=50*mm (for example, if the mother's solid were a Box of half lengths [25,25,25] then the replica's solid must be a box of half lengths [25,25,5]).

If the containing mother's solid is a tube of radius 50*mm and half Z length of 25*mm, `RepR` divides the mother tube into 5 cylinders (hence the solid associated with `pRepRLogical` must be a tube of radius 10*mm, and half Z length 25*mm); `repZ` divides the tube into 5 shorter cylinders (the solid associated with `pRepZLogical` must be a tube of radius 10*mm, and half Z length 5*mm); finally, `repPhi` divides the tube into 4 tube segments with full angle of 90 degrees (the solid associated with `pRepPhiLogical` must be a tube segment of radius 10*mm, half Z length 5*mm and delta phi of M_PI*0.5*rad).

No further volumes may be placed inside these replicas. To do so would result in intersecting boundaries due to the `r` replications.

### Parameterised Volumes

Parameterised Volumes are *repeated volumes* in the case in which the multiple copies of a volume can be different in size, solid type, or material. The solid's type, its dimensions, the material and the transformation matrix can all be parameterised in function of the copy number, both when a strong symmetry exist and when it does not. The user implements the desired parameterisation function and the program computes and updates automatically at run time the information associated to the Physical Volume.

An example of creating a parameterised volume (by dimension and position) exists in basic example B2b. The implementation is provided in the two classes `B2b::DetectorConstruction` and `B2b::ChamberParameterisation`.

To create a parameterised volume, one must first create its logical volume like `trackerChamberLV` below. Then one must create his own parameterisation class (*B2b::ChamberParameterisation*) and instantiate an object of this class (`chamberParam`). We will see how to create the parameterisation below.

Listing 4.3: An example of Parameterised volumes.

```
// Tracker segments

// An example of Parameterised volumes
// Dummy values for G4Tubs -- modified by parameterised volume

G4Tubs* chamberS
  = new G4Tubs("tracker",0, 100*cm, 100*cm, 0.*deg, 360.*deg);
fLogicChamber
  = new G4LogicalVolume(chamberS,fChamberMaterial,"Chamber",0,0,0);

G4double firstPosition = -trackerSize + chamberSpacing;
G4double firstLength   = trackerLength/10;
G4double lastLength    = trackerLength;

G4VPVParameterisation* chamberParam =
  new ChamberParameterisation(NbOfChambers,   // NoChambers
                              firstPosition,  // Z of center of first
                              chamberSpacing, // Z spacing of centers
                              chamberWidth,   // chamber width
                              firstLength,    // initial length
                              lastLength);    // final length

// dummy value : kZAxis -- modified by parameterised volume

new G4PVParameterised("Chamber",       // their name
                      fLogicChamber,   // their logical volume
                      trackerLV,       // Mother logical volume
                      kZAxis,          // Are placed along this axis
                      NbOfChambers,    // Number of chambers
                      chamberParam,    // The parametrisation
                      fCheckOverlaps); // checking overlaps
```

The general constructor is:

```
G4PVParameterised( const G4String&              pName,
                         G4LogicalVolume*       pCurrentLogical,
                         G4LogicalVolume*       pMotherLogical, // OR G4VPhysicalVolume*
                   const EAxis                  pAxis,
                   const G4int                  nReplicas,
                         G4VPVParameterisation* pParam,
                         G4bool                 pSurfChk=false )
```

Note that for a parameterised volume the user must always specify a mother volume. So the world volume can *never* be a parameterised volume, nor it can be sliced. The mother volume can be specified either as a physical or a logical volume.

pAxis specifies the tracking optimisation algorithm to apply: if a valid axis (the axis along which the parameterisation is performed) is specified, a simple one-dimensional voxelisation algorithm is applied; if "kUndefined" is specified instead, the default three-dimensional voxelisation algorithm applied for normal placements will be activated. In the latter case, more voxels will be generated, therefore a greater amount of memory will be consumed by the optimisation algorithm.

pSurfChk if true activates a check for overlaps with existing volumes or paramaterised instances.

The parameterisation mechanism associated to a parameterised volume is defined in the parameterisation class and its methods. Every parameterisation must create two methods:

- ComputeTransformation defines where one of the copies is placed,
- ComputeDimensions defines the size of one copy, and
- a constructor that initializes any member variables that are required.

An example is B2b::ChamberParameterisation that parameterises a series of tubes of different sizes

Listing 4.4: An example of Parameterised tubes of different sizes.

```
namespace B2b
{

class ChamberParameterisation : public G4VPVParameterisation
{
  ...
  void ComputeTransformation(const G4int       copyNo,
                             G4VPhysicalVolume *physVol) const;

  void ComputeDimensions(G4Tubs&             trackerLayer,
                         const G4int             copyNo,
                         const G4VPhysicalVolume *physVol) const;
  ...
}

}
```

These methods works as follows:

The ComputeTransformation method is called with a copy number for the instance of the parameterisation under consideration. It must compute the transformation for this copy, and set the physical volume to utilize this transformation:

```
void ChamberParameterisation::ComputeTransformation
(const G4int copyNo, G4VPhysicalVolume *physVol) const
{
  // Note: copyNo will start with zero!
  G4double Zposition = fStartZ + copyNo * fSpacing;
  G4ThreeVector origin(0,0,Zposition);
  physVol->SetTranslation(origin);
  physVol->SetRotation(0);
}
```

Note that the translation and rotation given in this scheme are those for the frame of coordinates (the *passive* method). They are **not** for the *active* method, in which the solid is rotated into the mother frame of coordinates.

Similarly the ComputeDimensions method is used to set the size of that copy.

```
void ChamberParameterisation::ComputeDimensions
(G4Tubs& trackerChamber, const G4int copyNo, const G4VPhysicalVolume*) const
{
  // Note: copyNo will start with zero!
  G4double rmax = fRmaxFirst + copyNo * fRmaxIncr;
  trackerChamber.SetInnerRadius(0);
  trackerChamber.SetOuterRadius(rmax);
```

(continues on next page)

(continued from previous page)

```
  trackerChamber.SetZHalfLength(fHalfWidth);
  trackerChamber.SetStartPhiAngle(0.*deg);
  trackerChamber.SetDeltaPhiAngle(360.*deg);
}
```

The user must ensure that the type of the first argument of this method (in this example `G4Tubs &`) corresponds to the type of object the user give to the logical volume of parameterised physical volume.

More advanced usage allows the user:

- to change the type of solid by creating a `ComputeSolid` method, or
- to change the material of the volume by creating a `ComputeMaterial` method. This method can also utilise information from a parent or other ancestor volume (see the Nested Parameterisation below.)

for the parameterisation.

Example `examples/extended/runAndEvent/RE02` shows a simple parameterisation by material. A more complex example is provided in `examples/extended/medical/DICOM`, where a phantom grid of cells is built using a parameterisation by material defined through a map.

**Note:** Currently for many cases it is not possible to add daughter volumes to a parameterised volume. Only parameterised volumes all of whose solids have the same size are allowed to contain daughter volumes. When the size or type of solid varies, adding daughters is not supported. So the full power of parameterised volumes can be used only for "leaf" volumes, which contain no other volumes.

**Note:** A hierarchy of volumes included in a parameterised volume cannot vary. Therefore, it is not possible to implement a parameterisation which can modify the hierarchy of volumes included inside a specific parameterised copy.

**Note:** For parameterisations of tubes or cons, where the starting `Phi` and its `DeltaPhi` angles vary, it is possible to optimise the regeneration of the trigonometric parameters of the shape, by invoking `SetStartPhiAngle(newPhi, false); SetDeltaPhiAngle (newDPhi)`, i.e. by specifying with `false` flag to skip the computation of the parameters which will be later on properly initialised with the call for `DeltaPhi`.

**Note:** Parameterisations of composed solids like Boolean, Reflected or Displaced solids are not recommended, given the complexity in handling transformations that this might imply, and limitations in making persistent representations (i.e. GDML) of the geometry itself.

**Note:** For multi-threaded applications, one must be careful in the implementation of the parameterisation functions for the geometrical objects being created in the parameterisation. In particular, when parameterising by the type of a solid, it is assumed that the solids being parameterised are being declared thread-local in the user's parameterisation class and allocated just once.

### Advanced parameterisations for 'nested' parameterised volumes

A different type of parameterisation enables a user to have the daughter's material also depend on the copy number of the parent when a parameterised volume (daughter) is located inside another (parent) repeated volume. The parent volume can be a replica, a parameterised volume, or a division if the key feature of modifying its contents is utilised. (Note: a 'nested' parameterisation inside a placement volume is not supported, because all copies of a placement volume must be identical at all levels.)

In such a " nested" parameterisation , the user must provide a `ComputeMaterial` method that utilises the new argument that represents the touchable history of the parent volume:

```
// Sample Parameterisation
class SampleNestedParameterisation : public G4VNestedParameterisation
{
 public:
   // .. other methods ...
    // Mandatory method, required and reason for this class
    virtual G4Material* ComputeMaterial(G4VPhysicalVolume *currentVol,
                                        const G4int no_lev,
                                        const G4VTouchable *parentTouch);
 private:
    G4Material *material1, *material2;
};
```

The implementation of the method can utilise any information from a parent or other ancestor volume of its parameterised physical volume, but typically it will use only the copy number:

```
G4Material*
SampleNestedParameterisation::ComputeMaterial(G4VPhysicalVolume *currentVol,
                                              const G4int no_lev,
                                              const G4VTouchable *parentTouchable)
{
   G4Material *material=0;

   // Get the information about the parent volume
   G4int no_parent= parentTouchable->GetReplicaNumber();
   G4int no_total= no_parent + no_lev;
   // A simple 'checkerboard' pattern of two materials
   if( no_total / 2 == 1 ) material= material1;
   else  material= material2;
   // Set the material to the current logical volume
   G4LogicalVolume* currentLogVol= currentVol->GetLogicalVolume();
   currentLogVol->SetMaterial( material );
   return material;
}
```

Nested parameterisations are suitable for the case of regular, 'voxel' geometries in which a large number of 'equal' volumes are required, and their only difference is in their material. By creating two (or more) levels of parameterised physical volumes it is possible to divide space, while requiring only limited additional memory for very fine-level optimisation. This provides fast navigation. Alternative implementations, taking into account the regular structure of such geometries in navigation are under study.

**Note:** You can also switch the colour of individual volumes by changing the vis attributes in your `ComputeMaterial` - see `examples//extended/medical/DICOM` or `examples/advanced/ICRP110_HumanPhantoms`.

**Note:** The number of parameterised volumes can become very large, in the 10's of millions for a medical phantom, for example. This can give the graphics system a headache. See *Visualization of a parameterised volume* for economical ways of visualising such parameterisations.

### Divisions of Volumes

Divisions in GEANT4 are *repeated volumes* and are implemented as a specialized type of parameterised volumes.

They serve to divide a volume into identical copies along one of its axes, providing the possibility to define an *offset*, and without the limitation that the daughters have to fill the mother volume as it is the case for the replicas. In the case, for example, of a tube divided along its radial axis, the copies are not strictly identical, but have increasing radii, although their widths are constant.

To divide a volume it will be necessary to provide:

1. the axis of division, and
2. either
    - the number of divisions (so that the width of each division will be automatically calculated), or
    - the division width (so that the number of divisions will be automatically calculated to fill as much of the mother as possible), or
    - both the number of divisions and the division width (this is especially designed for the case where the copies do not fully fill the mother).

An *offset* can be defined so that the first copy will start at some distance from the mother wall. The dividing copies will be then distributed to occupy the rest of the volume.

There are three constructors, corresponding to the three input possibilities described above:

- Giving only the number of divisions:

```
G4PVDivision( const G4String& pName,
                    G4LogicalVolume* pCurrentLogical,
                    G4LogicalVolume* pMotherLogical,
              const EAxis pAxis,
              const G4int nDivisions,
              const G4double offset )
```

- Giving only the division width:

```
G4PVDivision( const G4String& pName,
                    G4LogicalVolume* pCurrentLogical,
                    G4LogicalVolume* pMotherLogical,
              const EAxis pAxis,
              const G4double width,
              const G4double offset )
```

- Giving the number of divisions and the division width:

```
G4PVDivision( const G4String& pName,
                    G4LogicalVolume* pCurrentLogical,
                    G4LogicalVolume* pMotherLogical,
              const EAxis pAxis,
              const G4int nDivisions,
              const G4double width,
              const G4double offset )
```

where:

| | |
|---|---|
| `pName` | String identifier for the replicated volume |
| `pCurrentLogical` | The associated Logical Volume |
| `pMotherLogical` | The associated mother Logical Volume |
| `pAxis` | The axis along which the division is applied |
| `nDivisions` | The number of divisions |
| `width` | The width of a single division along the axis |
| `offset` | Possible offset associated to the mother along the axis of division |

The parameterisation is calculated automatically using the values provided in input. Therefore the dimensions of the solid associated with `pCurrentLogical` will not be used, but recomputed through the `G4VParameterisation::ComputeDimension()` method.

Since `G4VPVParameterisation` may have different `ComputeDimension()` methods for each solid type, the user must provide a solid that is of the same type as of the one associated to the mother volume.

As for any replica, the coordinate system of the divisions is related to the centre of each division for the Cartesian axis. For the radial axis, the coordinate system is the same of the mother volume. For the phi axis, the new coordinate system is rotated such that the X axis bisects the angle made by each wedge, and Z remains parallel to the mother's Z axis.

As divisions are parameterised volumes with constant dimensions, they may be placed inside other divisions, except in the case of divisions along the radial axis.

It is also possible to place other volumes inside a volume where a division is placed.

The list of volumes that currently support divisioning and the possible division axis are summarised below:

| | |
|---|---|
| `G4Box` | `kXAxis`, `kYAxis`, `kZAxis` |
| `G4Tubs` | `kRho`, `kPhi`, `kZAxis` |
| `G4Cons` | `kRho`, `kPhi`, `kZAxis` |
| `G4Trd` | `kXAxis`, `kYAxis`, `kZAxis` |
| `G4Para` | `kXAxis`, `kYAxis`, `kZAxis` |
| `G4Polycone` | `kRho`, `kPhi`, `kZAxis` |
| `G4Polyhedra` | `kRho`, `kPhi`, `kZAxis` (*) |

(*) - `G4Polyhedra`:

- `kPhi` - the number of divisions has to be the same as solid sides, (i.e. `numSides`), the width will *not* be taken into account.

In the case of division along `kRho` of `G4Cons`, `G4Polycone`, `G4Polyhedra`, if width is provided, it is taken as the width at the `-Z` radius; the width at other radii will be scaled to this one.

Examples are given below in listings Listing 4.3 and Listing 4.5.

Listing 4.5: An example of a box division along different axes, with or without offset.

```
G4Box* motherSolid = new G4Box("motherSolid", 0.5*m, 0.5*m, 0.5*m);
G4LogicalVolume* motherLog = new G4LogicalVolume(motherSolid, material, "mother",0,0,0);
G4Para* divSolid = new G4Para("divSolid", 0.512*m, 1.21*m, 1.43*m);
G4LogicalVolume* childLog = new G4LogicalVolume(divSolid, material, "child",0,0,0);

G4PVDivision divBox1("division along X giving nDiv",
                     childLog, motherLog, kXAxis, 5, 0.);

G4PVDivision divBox2("division along X giving width and offset",
                     childLog, motherLog, kXAxis, 0.1*m, 0.45*m);
```

(continues on next page)

(continued from previous page)

```
G4PVDivision divBox3("division along X giving nDiv, width and offset",
                     childLog, motherLog, kXAxis, 3, 0.1*m, 0.5*m);
```

- `divBox1` is a division of a box along its `X` axis in 5 equal copies. Each copy will have a dimension in meters of `[0.2, 1., 1.]`.
- `divBox2` is a division of the same box along its `X` axis with a width of `0.1` meters and an offset of `0.5` meters. As the mother dimension along `X` of `1` meter (`0.5*m` of halflength), the division will be sized in total `1 - 0.45 = 0.55` meters. Therefore, there's space for 5 copies, the first extending from `-0.05` to `0.05` meters in the mother's frame and the last from `0.35` to `0.45` meters.
- `divBox3` is a division of the same box along its `X` axis in 3 equal copies of width `0.1` meters and an offset of `0.5` meters. The first copy will extend from `0.` to `0.1` meters in the mother's frame and the last from `0.2` to `0.3` meters.

Listing 4.6: An example of division of a polycone.

```
G4double* zPlanem = new G4double[3];
          zPlanem[0]= -1.*m;
          zPlanem[1]= -0.25*m;
          zPlanem[2]=  1.*m;
G4double* rInnerm = new G4double[3];
          rInnerm[0]=0.;
          rInnerm[1]=0.1*m;
          rInnerm[2]=0.5*m;
G4double* rOuterm  = new G4double[3];
          rOuterm[0]=0.2*m;
          rOuterm[1]=0.4*m;
          rOuterm[2]=1.*m;
G4Polycone* motherSolid = new G4Polycone("motherSolid", 20.*deg, 180.*deg,
                                        3, zPlanem, rInnerm, rOuterm);
G4LogicalVolume* motherLog = new G4LogicalVolume(motherSolid, material, "mother",0,0,0);

G4double* zPlaned = new G4double[3];
          zPlaned[0]= -3.*m;
          zPlaned[1]= -0.*m;
          zPlaned[2]=  1.*m;
G4double* rInnerd = new G4double[3];
          rInnerd[0]=0.2;
          rInnerd[1]=0.4*m;
          rInnerd[2]=0.5*m;
G4double* rOuterd  = new G4double[3];
          rOuterd[0]=0.5*m;
          rOuterd[1]=0.8*m;
          rOuterd[2]=2.*m;
G4Polycone* divSolid = new G4Polycone("divSolid", 0.*deg, 10.*deg,
                                     3, zPlaned, rInnerd, rOuterd);
G4LogicalVolume* childLog = new G4LogicalVolume(divSolid, material, "child",0,0,0);

G4PVDivision divPconePhiW("division along phi giving width and offset",
                          childLog, motherLog, kPhi, 30.*deg, 60.*deg);

G4PVDivision divPconeZN("division along Z giving nDiv and offset",
                         childLog, motherLog, kZAxis, 2, 0.1*m);
```

- `divPconePhiW` is a division of a polycone along its `phi` axis in equal copies of width 30 degrees with an offset of 60 degrees. As the mother extends from 0 to 180 degrees, there's space for 4 copies. All the copies have a starting angle of 20 degrees (as for the mother) and a `phi` extension of 30 degrees. They are rotated around the `Z` axis by 60 and 30 degrees, so that the first copy will extend from 80 to 110 and the last from 170 to 200 degrees.
- `divPconeZN` is a division of the same polycone along its `Z` axis. As the mother polycone has two sections, it will be divided in two one-section polycones, the first one extending from -1 to -0.25 meters, the second from

-0.25 to 1 meters. Although specified, the offset will not be used.

## Replicated Slices

A special kind of divided volume is represented by G4ReplicatedSlice, a division allowing for gaps inbetween divided volumes.

Three constructors, corresponding to three input possibilities are provided:

- Giving only the number of divisions:

```
G4ReplicatedSlice( const G4String& pName,
                         G4LogicalVolume* pCurrentLogical,
                         G4LogicalVolume* pMotherLogical,
                   const EAxis pAxis,
                   const G4int nDivisions,
                   const G4double half_gap,
                   const G4double offset )
```

- Giving only the division width:

```
G4ReplicatedSlice( const G4String& pName,
                         G4LogicalVolume* pCurrentLogical,
                         G4LogicalVolume* pMotherLogical,
                   const EAxis pAxis,
                   const G4double width,
                   const G4double half_gap,
                   const G4double offset )
```

- Giving the number of divisions and the division width:

```
G4ReplicatedSlice( const G4String& pName,
                         G4LogicalVolume* pCurrentLogical,
                         G4LogicalVolume* pMotherLogical,
                   const EAxis pAxis,
                   const G4int nDivisions,
                   const G4double width,
                   const G4double half_gap,
                   const G4double offset )
```

where:

| | |
|---|---|
| pName | String identifier for the replicated volume |
| pCurrentLogical | The associated Logical Volume |
| pMotherLogical | The associated mother Logical Volume |
| pAxis | The axis along which the division is applied |
| nDivisions | The number of divisions |
| width | The width of a single division along the axis |
| half_gap | The half width of the gap to be considered inbetween division slices |
| offset | Possible offset associated to the mother along the axis of division |

As for G4PVDivision, the parameterisation is calculated automatically using the values provided in input.

### 4.1.5 Touchables: Uniquely Identifying a Volume

#### Introduction to Touchables

A *touchable* for a volume serves the purpose of providing a unique identification for a detector element. This can be useful for description of the geometry alternative to the one used by the GEANT4 tracking system, such as a Sensitive Detectors based read-out geometry, or a parameterised geometry for fast Monte Carlo. In order to create a *touchable volume*, several techniques can be implemented: for example, in GEANT4 touchables are implemented as solids associated to a transformation-matrix in the global reference system, or as a hierarchy of physical volumes up to the root of the geometrical tree.

A touchable is a geometrical entity (volume or solid) which has a unique placement in a detector description. It is represented by an abstract base class which can be implemented in a variety of ways. Each way must provide the capabilities of obtaining the transformation and solid that is described by the touchable.

#### What can a Touchable do?

All `G4VTouchable` implementations must respond to the two following "requests", where in all cases, by `depth` it is meant the number of levels *up* in the tree to be considered (the default and current one is `0`):

1. `GetTranslation(depth)`
2. `GetRotation(depth)`

that return the components of the volume's transformation.

Additional capabilities are available from implementations with more information. These have a default implementation that causes an exception.

Several capabilities are available from touchables with physical volumes:

1. `GetSolid(depth)` gives the solid associated to the touchable.
2. `GetVolume(depth)` gives the physical volume.
3. `GetReplicaNumber(depth)` or `GetCopyNumber(depth)` which return the copy number of the physical volume (replicated or not).

Touchables that store volume hierarchy (history) have the whole stack of parent volumes available. Thus it is possible to add a little more state in order to extend its functionality. We add a "pointer" to a level and a member function to move the level in this stack. Then calling the above member functions for another level the information for that level can be retrieved.

The top of the history tree is, by convention, the world volume.

1. `GetHistoryDepth()` gives the depth of the history tree.
2. `MoveUpHistory(num)` moves the current pointer inside the touchable to point `num` levels up the history tree. Thus, e.g., calling it with `num=1` will cause the internal pointer to move to the mother of the current volume.

   **Warning:** this function changes the state of the touchable and can cause errors in tracking if applied to Pre/Post step touchables.

These methods are valid only for the *touchable-history* type, as specified also below.

An update method, with different arguments is available, so that the information in a touchable can be updated:

1. `UpdateYourself(vol, history)` takes a physical volume pointer and can additionally take a `NavigationHistory` pointer.

### Touchable history holds stack of geometry data

As shown in Sections *Logical Volumes* and *Physical Volumes*, a logical volume represents unpositioned detector elements, and a physical volume can represent multiple detector elements. On the other hand, touchables provide a unique identification for a detector element. In particular, the GEANT4 transportation process and the tracking system exploit touchables as implemented in `G4TouchableHistory`. The touchable history is the minimal set of information required to specify the full genealogy of a given physical volume (up to the root of the geometrical tree). These touchable volumes are made available to the user at every step of the GEANT4 tracking in `G4VUserSteppingAction`.

To create/access a `G4TouchableHistory` the user must message `G4Navigator` which provides the method `CreateTouchableHistoryHandle()`:

```
G4TouchableHistoryHandle CreateTouchableHistoryHandle() const;
```

this will return a handle to the touchable.

The methods that differentiate the touchable-history from other touchables (since they have meaning only for this type…), are:

```
G4int GetHistoryDepth()  const;
G4int MoveUpHistory( G4int num_levels = 1 );
```

The first method is used to find out how many levels deep in the geometry tree the current volume is. The second method asks the touchable to eliminate its deepest level.

As mentioned above, `MoveUpHistory(num)` significantly modifies the state of a touchable.

## 4.1.6 Creating an Assembly of Volumes

`G4AssemblyVolume` is a helper class which allows several logical volumes to be combined together in an arbitrary way in 3D space. The result is a placement of a normal logical volume, but where final physical volumes are many.

However, an *assembly* volume does not act as a real mother volume, being an envelope for its daughter volumes. Its role is over at the time the placement of the logical assembly volume is done. The physical volume objects become independent copies of each of the assembled logical volumes.

This class is particularly useful when there is a need to create a regular pattern in space of a complex component which consists of different shapes and can't be obtained by using replicated volumes or parametrised volumes (see also Fig. 4.2. Careful usage of `G4AssemblyVolume` must be considered though, in order to avoid cases of "proliferation" of physical volumes all placed in the same mother.

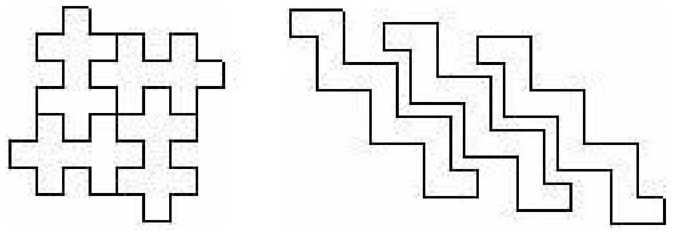


Fig. 4.2: Examples of *assembly* of volumes.

### Filling an assembly volume with its "daughters"

Participating logical volumes are represented as a triplet of <logical volume, translation, rotation> (`G4AssemblyTriplet` class).

The adopted approach is to place each participating logical volume with respect to the assembly's coordinate system, according to the specified translation and rotation.

### Assembly volume placement

An assembly volume object is composed of a set of logical volumes; imprints of it can be made inside a mother logical volume.

Since the assembly volume class generates physical volumes during each imprint, the user has no way to specify identifiers for these. An internal counting mechanism is used to compose uniquely the names of the physical volumes created by the invoked `MakeImprint(...)` method(s).

The name for each of the physical volume is generated with the following format:

```
av_WWW_impr_XXX_YYY_ZZZ
```

where:

```
WWW - assembly volume instance number
XXX - assembly volume imprint number
YYY - the name of the placed logical volume
ZZZ - the logical volume index inside the assembly volume
```

It is however possible to access the constituent physical volumes of an assembly and eventually customise ID and copy-number.

The setting of the copy-numbers can be complex, depending on how complex is the structure being built. Each assembly (`G4AssemblyVolume`) instance gets automatically assigned a number, `assemblyID`, which starts from zero and gets incremented based on the number of imprints being made. Each assembly is being stored in a `G4AssemblyStore` and can always been retrieved at any time. `G4AssemblyVolume` allows to define a *base copy-number* for each imprint (call to `MakeImprint()`), by specifying it as a parameter, `copyNumBase`, which is set to zero by default. The computation of the effective copy-number of each volume in the assembly is done using such parameter, i.e. based on the number of triplets (number of volumes added in the assembly), each volume copy number is assigned as:

```
numberOfDaughters + i
```

where `i` goes from zero to the number of volumes in the assembly; `numberOfDaughters` is either set to the number of daughter volumes in the mother where the assembly must be placed (if `copyNumBase` is zero, i.e. not being specified at the time the imprint is made), *or* the specified `copyNumBase`.

In case the assembly includes another assembly inside, the call to `makeImprint()` is made recursively, and the *base copy-number* in this case is being set to:

```
i*100+copyNumBase
```

so, shifted by 100 times the index of the triplet in the original assembly.

### Destruction of an assembly volume

At destruction all the generated physical volumes and associated rotation matrices of the imprints will be destroyed. A list of physical volumes created by `MakeImprint()` method is kept, in order to be able to cleanup the objects when not needed anymore. This requires the user to keep the assembly objects in memory during the whole job or during the life-time of the `G4Navigator`, logical volume store and physical volume store may keep pointers to physical volumes generated by the assembly volume.

The `MakeImprint()` method will operate correctly also on transformations including reflections and can be applied also to recursive assemblies (i.e., it is possible to generate imprints of assemblies including other assemblies). Giving `true` as the last argument of the `MakeImprint()` method, it is possible to activate the volumes overlap check for the assembly's constituents (the default is `false`).

Each assembly structure is registered at construction in a specialised store, `G4AssemblyStore`, which can then be used to identify all structures defined in a geometry setup, as well as the volumes belonging to each imprint.

At destruction of a `G4AssemblyVolume`, all its generated physical volumes and rotation matrices will be automatically freed.

### Example

This example shows how to use the `G4AssemblyVolume` class. It implements a layered detector where each layer consists of 4 plates.

In the code below, at first the world volume is defined, then solid and logical volume for the plate are created, followed by the definition of the assembly volume for the layer.

The assembly volume for the layer is then filled by the plates in the same way as normal physical volumes are placed inside a mother volume.

Finally the layers are placed inside the world volume as the imprints of the assembly volume (see Listing 4.7).

Listing 4.7: An example of usage of the `G4AssemblyVolume` class.

```
static unsigned int layers = 5;

void TstVADetectorConstruction::ConstructAssembly()
{
  // Define world volume
  G4Box* WorldBox = new G4Box( "WBox", worldX/2., worldY/2., worldZ/2. );
  G4LogicalVolume*   worldLV   = new G4LogicalVolume( WorldBox, selectedMaterial,
                                                      "WLog", 0, 0, 0 );
  G4VPhysicalVolume* worldVol = new G4PVPlacement( 0, G4ThreeVector(), "WPhys",worldLV,
                                                    0, false, 0 );

  // Define a plate
  G4Box* PlateBox = new G4Box( "PlateBox", plateX/2., plateY/2., plateZ/2. );
  G4LogicalVolume* plateLV = new G4LogicalVolume( PlateBox, Pb, "PlateLV", 0, 0, 0 );

  // Define one layer as one assembly volume
  G4AssemblyVolume* assemblyDetector = new G4AssemblyVolume();

  // Rotation and translation of a plate inside the assembly
  G4RotationMatrix Ra;
  G4ThreeVector Ta;
  G4Transform3D Tr;

  // Rotation of the assembly inside the world
  G4RotationMatrix Rm;

  // Fill the assembly by the plates
  Ta.setX( caloX/4. ); Ta.setY( caloY/4. ); Ta.setZ( 0. );
```

(continues on next page)

(continued from previous page)

```
  Tr = G4Transform3D(Ra,Ta);
  assemblyDetector->AddPlacedVolume( plateLV, Tr );

  Ta.setX( -1*caloX/4. ); Ta.setY( caloY/4. ); Ta.setZ( 0. );
  Tr = G4Transform3D(Ra,Ta);
  assemblyDetector->AddPlacedVolume( plateLV, Tr );

  Ta.setX( -1*caloX/4. ); Ta.setY( -1*caloY/4. ); Ta.setZ( 0. );
  Tr = G4Transform3D(Ra,Ta);
  assemblyDetector->AddPlacedVolume( plateLV, Tr );

  Ta.setX( caloX/4. ); Ta.setY( -1*caloY/4. ); Ta.setZ( 0. );
  Tr = G4Transform3D(Ra,Ta);
  assemblyDetector->AddPlacedVolume( plateLV, Tr );

  // Now instantiate the layers
  for( unsigned int i = 0; i < layers; i++ )
  {
    // Translation of the assembly inside the world
    G4ThreeVector Tm( 0,0,i*(caloZ + caloCaloOffset) - firstCaloPos );
    Tr = G4Transform3D(Rm,Tm);
    assemblyDetector->MakeImprint( worldLV, Tr );
  }
}
```

The resulting detector will look as in Fig. 4.3.

## 4.1.7 Reflecting Hierarchies of Volumes

Hierarchies of placed or replicated volumes can be reflected by means of the `G4ReflectionFactory` class and `G4ReflectedSolid`, which implements a solid that has been shifted from its original reference frame to a new 'reflected' one. The reflection transformation is applied as a decomposition into rotation and translation transformations.

The factory is a singleton object which provides the following methods:

```
G4PhysicalVolumesPair Place(const G4Transform3D&   transform3D,
                            const G4String&        name,
                                  G4LogicalVolume* LV,
                                  G4LogicalVolume* motherLV,
                                  G4bool           isMany,
                                  G4int            copyNo,
                                  G4bool           surfCheck=false)

G4PhysicalVolumesPair Replicate(const G4String&        name,
                                      G4LogicalVolume* LV,
                                      G4LogicalVolume* motherLV,
                                      EAxis            axis,
                                      G4int            nofReplicas,
                                      G4double         width,
                                      G4double         offset=0)

G4PhysicalVolumesPair Divide(const G4String&        name,
                                   G4LogicalVolume* LV,
                                   G4LogicalVolume* motherLV,
                                   EAxis            axis,
                                   G4int            nofDivisions,
                                   G4double         width,
                                   G4double         offset);
```

The method `Place()` used for placements, evaluates the passed transformation. In case the transformation contains a reflection, the factory will act as follows:

1. Performs the transformation decomposition.

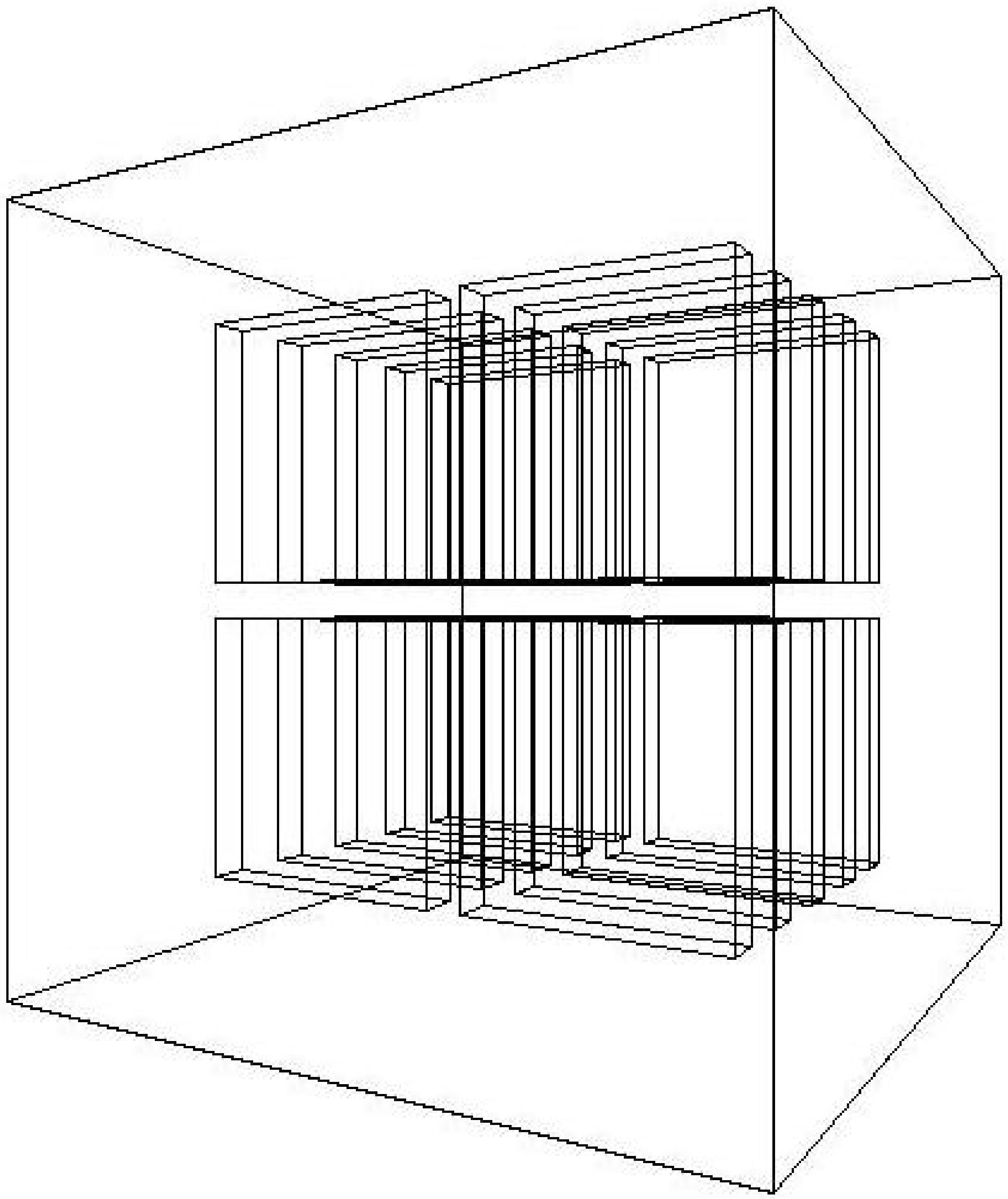


Fig. 4.3: The geometry corresponding to the Listing 4.7.

2. Creates a new reflected solid and logical volume, or retrieves them from a map if the reflected object was already created.
3. Transforms the daughters (if any) and place them in the given mother.

If successful, the result is a pair of physical volumes, where the second physical volume is a placement in a reflected mother. Optionally, it is also possible to force the overlaps check at the time of placement, by activating the `surfCheck` flag.

The method `Replicate()` creates replicas in the given mother. If successful, the result is a pair of physical volumes, where the second physical volume is a replica in a reflected mother.

The method `Divide()` creates divisions in the given mother. If successful, the result is a pair of physical volumes, where the second physical volume is a division in a reflected mother. There exists also two more variants of this method which may specify or not width or number of divisions.

**Note:** In order to reflect hierarchies containing divided volumes, it is necessary to explicitly instantiate a concrete *division* factory -before- applying the actual reflection: (i.e. - `G4PVDivisionFactory::GetInstance();`).

**Note:** Reflection of generic parameterised volumes is currently not possible.

Listing 4.8: An example of usage of the G4ReflectionFactory class.

```
#include "G4ReflectionFactory.hh"

// Calor placement with rotation

G4double calThickness = 100*cm;
G4double Xpos = calThickness*1.5;
G4RotationMatrix* rotD3 = new G4RotationMatrix();
rotD3->rotateY(10.*deg);

G4VPhysicalVolume* physiCalor =
    new G4PVPlacement(rotD3,                     // rotation
                      G4ThreeVector(Xpos,0.,0.), // at (Xpos,0,0)
                      logicCalor,     // its logical volume (defined elsewhere)
                      "Calorimeter",  // its name
                      logicHall,      // its mother volume (defined elsewhere)
                      false,          // no boolean operation
                      0);             // copy number

// Calor reflection with rotation
//
G4Translate3D translation(-Xpos, 0., 0.);
G4Transform3D rotation = G4Rotate3D(*rotD3);
G4ReflectX3D  reflection;
G4Transform3D transform = translation*rotation*reflection;

G4ReflectionFactory::Instance()
         ->Place(transform,     // the transformation with reflection
                 "Calorimeter", // the actual name
                 logicCalor,    // the logical volume
                 logicHall,     // the mother volume
                 false,         // no boolean operation
                 1,             // copy number
                 false);        // no overlap check triggered

// Replicate layers
//
G4ReflectionFactory::Instance()
         ->Replicate("Layer",    // layer name
```

(continues on next page)

(continued from previous page)

```
                    logicLayer, // layer logical volume (defined elsewhere)
                    logicCalor, // its mother
                    kXAxis,     // axis of replication
                    5,          // number of replica
                    20*cm);     // width of replica
```

## 4.1.8 The Geometry Navigator

Navigation through the geometry at tracking time is implemented by the class `G4Navigator`. The navigator is used to locate points in the geometry and compute distances to geometry boundaries. At tracking time, the navigator is intended to be the only point of interaction with tracking.

Internally, the G4Navigator has several private helper/utility classes:

- **G4NavigationHistory** - stores the compounded transformations, replication/parameterisation information, and volume pointers at each level of the hierarchy to the current location. The volume types at each level are also stored - whether normal (placement), replicated or parameterised.
- **G4NormalNavigation** - provides location & distance computation functions for geometries containing 'placement' volumes, with no voxels.
- **G4VoxelNavigation** - provides location and distance computation functions for geometries containing 'placement' physical volumes with voxels. Internally a stack of voxel information is maintained. Private functions allow for isotropic distance computation to voxel boundaries and for computation of the 'next voxel' in a specified direction.
- **G4ParameterisedNavigation** - provides location and distance computation functions for geometries containing parameterised volumes with voxels. Voxel information is maintained similarly to `G4VoxelNavigation`, but computation can also be simpler by adopting voxels to be one level deep only (*unrefined*, or 1D optimisation)
- **G4ReplicaNavigation** - provides location and distance computation functions for replicated volumes.
- **G4RegularNavigation** - provides location and distance computation functions for fast navigation in volumes containing a regular parameterisation. If two contiguous voxels have the same material, navigation does not stop at the surface.

In addition, the navigator maintains a set of flags for exiting/entry optimisation. A navigator is not a singleton class; this is mainly to allow a design extension in future (e.g. geometrical event biasing).

### Navigation and Tracking

The main functions required for tracking in the geometry are described below. Additional functions are provided to return the net transformation of volumes and for the creation of touchables. None of the functions implicitly requires that the geometry be described hierarchically.

- **SetWorldVolume()**
  Sets the first volume in the hierarchy. It must be unrotated and untranslated from the origin.
- **LocateGlobalPointAndSetup()**
  Locates the volume containing the specified global point. This involves a traverse of the hierarchy, requiring the computation of compound transformations, testing replicated and parameterised volumes (etc). To improve efficiency this search may be performed relative to the last, and this is the recommended way of calling the function. A 'relative' search may be used for the first call of the function which will result in the search defaulting to a search from the root node of the hierarchy. Searches may also be performed using a `G4TouchableHistory`.
- **LocateGlobalPointAndUpdateTouchableHandle()**
  First, search the geometrical hierarchy like the above method `LocateGlobalPointAndSetup()`. Then use the volume found and its navigation history to update the touchable.
- **ComputeStep()**
  Computes the distance to the next boundary intersected along the specified unit direction from a specified point. The point must be have been located prior to calling `ComputeStep()`.

When calling `ComputeStep()`, a proposed physics step is passed. If it can be determined that the first intersection lies at or beyond that distance then `kInfinity` is returned. In any case, if the returned step is greater than the physics step, the physics step must be taken.

- **SetGeometricallyLimitedStep()**
  Informs the navigator that the last computed step was taken in its entirety. This enables entering/exiting optimisation, and should be called prior to calling `LocateGlobalPointAndSetup()`.
- **CreateTouchableHistory()**
  Creates a `G4TouchableHistory` object, for which the caller has deletion responsibility. The 'touchable' volume is the volume returned by the last Locate operation. The object includes a copy of the current NavigationHistory, enabling the efficient relocation of points in/close to the current volume in the hierarchy.

As stated previously, the navigator makes use of utility classes to perform location and step computation functions. The different navigation utilities manipulate the `G4NavigationHistory` object.

In `LocateGlobalPointAndSetup()` the process of locating a point breaks down into three main stages - optimisation, determination that the point is contained with a subtree (mother and daughters), and determination of the actual containing daughter. The latter two can be thought of as scanning first 'up' the hierarchy until a volume that is guaranteed to contain the point is found, and then scanning 'down' until the actual volume that contains the point is found.

In `ComputeStep()` three types of computation are treated depending on the current containing volume:

- The volume contains normal (placement) daughters (or none)
- The volume contains a single parameterised volume object, representing many volumes
- The volume is a replica and contains normal (placement) daughters

### Using the navigator to locate points

More than one navigator object can be created inside an application; these navigators can act independently for different purposes. The main navigator which is *activated* automatically at the startup of a simulation program is the navigator used for the *tracking* and attached the world volume of the main tracking (or *mass*) geometry.

The navigator for tracking can be retrieved at any state of the application by messaging the `G4TransportationManager`:

```
G4Navigator* tracking_navigator =
  G4TransportationManager::GetInstance()->GetNavigatorForTracking();
```

This also allows to retrieve at any time a pointer to the world volume assigned for tracking:

```
G4VPhysicalVolume* tracking_world = tracking_navigator->GetWorldVolume();
```

The navigator for tracking also retains all the information of the current history of volumes traversed at a precise moment of the tracking during a run. Therefore, if the navigator for tracking is used during tracking for locating a generic point in the tree of volumes, the actual particle gets also -relocated- in the specified position and tracking will be of course affected !

In order to avoid the problem above and provide information about location of a point without affecting the tracking, it is suggested to either use an alternative `G4Navigator` object (which can then be assigned to the world-volume), or access the information through the step.

If the user instantiates an alternative `G4Navigator`, ownership is retained by the user's code, and the navigator object should be deleted by that code.

### Using the 'step' to retrieve geometrical information

During the tracking run, geometrical information can be retrieved through the touchable handle associated to the current step. For example, to identify the exact copy-number of a specific physical volume in the mass geometry, one should do the following:

```
// Given the pointer to the step object ...
//
G4Step* aStep = ..;

// ... retrieve the 'pre-step' point
//
G4StepPoint* preStepPoint = aStep->GetPreStepPoint();

// ... retrieve a touchable handle and access to the information
//
G4TouchableHandle theTouchable = preStepPoint->GetTouchableHandle();
G4int copyNo = theTouchable->GetCopyNumber();
G4int motherCopyNo = theTouchable->GetCopyNumber(1);
```

To determine the exact position in global coordinates in the mass geometry and convert to local coordinates (local to the current volume):

```
G4ThreeVector worldPosition = preStepPoint->GetPosition();
G4ThreeVector localPosition = theTouchable->GetHistory()->
              GetTopTransform().TransformPoint(worldPosition);
```

### Using an alternative navigator to locate points

In order to know (when in the `idle` state of the application) in which physical volume a given point is located in the detector geometry, it is necessary to create an alternative navigator object first and assign it to the world volume:

```
G4Navigator* aNavigator = new G4Navigator();
aNavigator->SetWorldVolume(worldVolumePointer);
```

Then, locate the point `myPoint` (defined in global coordinates), retrieve a *touchable handle* and do whatever you need with it:

```
aNavigator->LocateGlobalPointAndSetup(myPoint);
G4TouchableHistoryHandle aTouchable =
     aNavigator->CreateTouchableHistoryHandle();

   // Do whatever you need with it ...
   // ... convert point in local coordinates (local to the current volume)
   //
   G4ThreeVector localPosition = aTouchable->GetHistory()->
                 GetTopTransform().TransformPoint(myPoint);

   // ... convert back to global coordinates system
   G4ThreeVector globalPosition = aTouchable->GetHistory()->
                 GetTopTransform().Inverse().TransformPoint(localPosition);
```

If outside of the tracking run and given a generic local position (local to a given volume in the geometry tree), it is -not- possible to determine a priori its global position and convert it to the global coordinates system. The reason for this is rather simple, nobody can guarantee that the given (local) point is located in the right -copy- of the physical volume ! In order to retrieve this information, some extra knowledge related to the absolute position of the physical volume is required first, i.e. one should first determine a global point belonging to that volume, eventually making a dedicated scan of the geometry tree through a dedicated `G4Navigator` object and then apply the method above after having created the touchable for it.

### Navigation in parallel geometries

Since release 8.2 of GEANT4, it is possible to define geometry trees which are `parallel` to the tracking geometry and having them assigned to navigator objects that transparently communicate in sync with the normal tracking geometry.

Parallel geometries can be defined for several uses (fast shower parameterisation, geometrical biasing, particle scoring, readout geometries, etc …) and can *overlap* with the mass geometry defined for the tracking. The `parallel` transportation will be activated only after the registration of the parallel geometry in the detector description setup; see Section *Parallel Geometries* for how to define a parallel geometry and register it to the run-manager.

The `G4TransportationManager` provides all the utilities to verify, retrieve and activate the navigators associated to the various parallel geometries defined.

### Fast navigation in regular patterned geometries and phantoms

Since release 9.1 of GEANT4, a specialised navigation algorithm has been introduced to allow for optimal memory use and extremely efficient navigation in geometries represented by a regular pattern of volumes and particularly three-dimensional grids of boxes. A typical application of this kind is the case of DICOM phantoms for medical physics studies.

The class `G4RegularNavigation` is used and automatically activated when such geometries are defined. It is required to the user to implement a parameterisation of the kind `G4PhantomParameterisation` and place the parameterised volume containing it in a container volume, so that all cells in the three-dimensional grid (*voxels*) completely fill the container volume. This way the location of a point inside a voxel can be done in a fast way, transforming the position to the coordinate system of the container volume and doing a simple calculation of the kind:

```
copyNo_x = (localPoint.x()+fVoxelHalfX*fNoVoxelX)/(fVoxelHalfX*2.)
```

where `fVoxelHalfX` is the half dimension of the voxel along `X` and `fNoVoxelX` is the number of voxels in the `X` dimension. Voxel `0` will be the one closest to the corner `(fVoxelHalfX*fNoVoxelX, fVoxelHalfY*fNoVoxelY, fVoxelHalfZ*fNoVoxelZ)`.

Having the voxels filling completely the container volume allows to avoid the lengthy computation of `ComputeStep()` and `ComputeSafety` methods required in the traditional navigation algorithm. In this case, when a track is inside the parent volume, it has always to be inside one of the voxels and it will be only necessary to calculate the distance to the walls of the current voxel.

### Skipping borders of voxels with same material

Another speed optimisation can be provided by skipping the frontiers of two voxels which the same material assigned, so that bigger steps can be done. This optimisation may be not very useful when the number of materials is very big (in which case the probability of having contiguous voxels with same material is reduced), or when the physical step is small compared to the voxel dimensions (very often the case of electrons). The optimisation can be switched off in such cases, by invoking the following method with argument `skip = 0`:

### Phantoms with only one material

If you want to describe a phantom of a unique material, you may spare some memory by not filling the set of indices of materials of each voxel. If the method `SetMaterialIndices()` is not invoked, the index for all voxels will be 0, that is the first (and unique) material in your list.

```
G4RegularParameterisation::SetSkipEqualMaterials( G4bool skip );
```

### Example

To use the specialised navigation, it is required to first create an object of type `G4PhantomParameterisation`:

```
G4PhantomParameterisation* param = new G4PhantomParameterisation();
```

Then, fill it with the all the necessary data:

```
// Voxel dimensions in the three dimensions
//
G4double halfX = ...;
G4double halfY = ...;
G4double halfZ = ...;
param->SetVoxelDimensions( halfX, halfY, halfZ );

// Number of voxels in the three dimensions
//
G4int nVoxelX = ...;
G4int nVoxelY = ...;
G4int nVoxelZ = ...;
param->SetNoVoxel( nVoxelX, nVoxelY, nVoxelZ );

// Vector of materials of the voxels
//
std::vector < G4Material* > theMaterials;
theMaterials.push_back( new G4Material( ...
theMaterials.push_back( new G4Material( ...
param->SetMaterials( theMaterials );

// List of material indices
// For each voxel it is a number that correspond to the index of its
// material in the vector of materials defined above;
//
size_t* mateIDs = new size_t[nVoxelX*nVoxelY*nVoxelZ];
mateIDs[0] = n0;
mateIDs[1] = n1;
...
param->SetMaterialIndices( mateIDs );
```

Then, define the volume that contains all the voxels:

```
G4Box* cont_solid = new G4Box("PhantomContainer",nVoxelX*halfX.,nVoxelY*halfY.,nVoxelZ*halfZ);
G4LogicalVolume* cont_logic =
  new G4LogicalVolume( cont_solid,
           matePatient,        // material is not relevant here...
           "PhantomContainer",
           0, 0, 0 );
G4VPhysicalVolume * cont_phys =
  new G4PVPlacement(rotm,                   // rotation
            pos,                    // translation
            cont_logic,             // logical volume
            "PhantomContainer",     // name
            world_logic,            // mother volume
```

(continues on next page)

(continued from previous page)

```
            false,                  // No op. bool.
            1);                     // Copy number
```

The physical volume should be assigned as the container volume of the parameterisation:

```
param->BuildContainerSolid(cont_phys);

// Assure that the voxels are completely filling the container volume
//
param->CheckVoxelsFillContainer( cont_solid->GetXHalfLength(),
                                    cont_solid->GetyHalfLength(),
                                    cont_solid->GetzHalfLength() );

// The parameterised volume which uses this parameterisation is placed
// in the container logical volume
//
G4PVParameterised * patient_phys =
  new G4PVParameterised("Patient",               // name
                        patient_logic,           // logical volume
                        cont_logic,              // mother volume
            kXAxis,                  // optimisation hint
                        nVoxelX*nVoxelY*nVoxelZ, // number of voxels
                        param);                  // parameterisation

// Indicate that this physical volume is having a regular structure
//
patient_phys->SetRegularStructureId(1);
```

An example showing the application of the optimised navigation algorithm for phantoms geometries is available in `examples/extended/medical/DICOM`. It implements a real application for reading `DICOM` images and convert them to GEANT4 geometries with defined materials and densities, allowing for different implementation solutions to be chosen (non-optimised, classical 3D optimisation, nested parameterisations and use of `G4PhantomParameterisation`).

### Run-time commands

When running in *verbose* mode (i.e. the default, `G4VERBOSE` set while installing the GEANT4 kernel libraries), the navigator provides a few commands to control its behavior. It is possible to select different verbosity levels (up to 5), with the command:

```
geometry/navigator/verbose [verbose_level]
```

or to force the navigator to run in *check* mode:

```
geometry/navigator/check_mode [true/false]
```

The latter will force more strict and less tolerant checks in step/safety computation to verify the correctness of the solids' response in the geometry.

By combining *check_mode* with verbosity level-1, additional verbosity checks on the response from the solids can be activated.

### Setting Geometry Tolerance to be relative

The tolerance value defining the accuracy of tracking on the surfaces is by default set to a reasonably small value of *10E-9 mm*. Such accuracy may be however redundant for use on simulation of detectors of big size or macroscopic dimensions. Since release 9.0, it is possible to specify the surface tolerance to be relative to the extent of the world volume defined for containing the geometry setup.

The class `G4GeometryManager` can be used to activate the computation of the surface tolerance to be relative to the geometry setup which has been defined. It can be done this way:

```
G4GeometryManager::GetInstance()->SetWorldMaximumExtent(WorldExtent);
```

where, `WorldExtent` is the actual maximum extent of the world volume used for placing the whole geometry setup.

Such call to `G4GeometryManager` must be done **before** defining any geometrical component of the setup (solid shape or volume), and can be done only **once**!

The class `G4GeometryTolerance` is to be used for retrieving the actual values defined for tolerances, surface (Cartesian), angular or radial respectively:

```
G4GeometryTolerance::GetInstance()->GetSurfaceTolerance();
G4GeometryTolerance::GetInstance()->GetAngularTolerance();
G4GeometryTolerance::GetInstance()->GetRadialTolerance();
```

## 4.1.9 Converting Geometries from Geant3.21

### Approach

**G3toG4** is the GEANT4 facility to convert GEANT 3.21 geometries into GEANT4. This is done in two stages:

1. The user supplies a GEANT 3.21 RZ-file (.rz) containing the initialization data structures. An executable `rztog4` reads this file and produces an ASCII *call list* file containing instructions on how to build the geometry. The source code of `rztog4` is FORTRAN.
2. A call list interpreter (`G4BuildGeom.cc`) reads these instructions and builds the geometry in the user's client code for GEANT4.

### Importing converted geometries into GEANT4

Two examples of how to use the call list interpreter are supplied in the directory `examples/extended/g3tog4`:

1. `cltog4` is a simple example which simply invokes the call list interpreter method `G4BuildGeom` from the `G3toG4DetectorConstruction` class, builds the geometry and exits.
2. `clGeometry`, is more complete and is patterned as for the basic GEANT4 examples. It also invokes the call list interpreter, but in addition, allows the geometry to be visualized and particles to be tracked.

To compile and build the G3toG4 libraries, you need to have enabled `GEANT4_USE_G3TOG4` at the build configuration of GEANT4. The G3toG4 libraries are not built by default.

### Current Status

The package has been tested with the geometries from experiments like: BaBar, CMS, Atlas, Alice, Zeus, L3, and Opal.

Here is a comprehensive list of features supported and not supported or implemented in the current version of the package:

- Supported shapes: all GEANT 3.21 shapes except for `GTRA`, `CTUB`.
- `PGON`, `PCON` are built using the *specific* solids `G4Polycone` and `G4Polyhedra`.
- GEANT 3.21 `MANY` feature is only partially supported. `MANY` positions are resolved in the `G3toG4MANY()` function, which has to be processed before `G3toG4BuildTree()` (it is not called by default). In order to resolve `MANY`, the user code has to provide additional info using `G4gsbool(G4String volName, G4String manyVolName)` function for all the overlapping volumes. Daughters of overlapping volumes are then resolved automatically and should not be specified via `Gsbool`.
  **Limitation**: a volume with a `MANY` position can have only this one position; if more than one position is needed a new volume has to be defined (`gsvolu()`) for each position.
- `GSDV*` routines for dividing volumes are implemented, using `G4PVReplicas`, for shapes:
  - `BOX`, `TUBE`, `TUBS`, `PARA` - all axes;
  - `CONE`, `CONS` - axes 2, 3;
  - `TRD1`, `TRD2`, `TRAP` - axis 3;
  - `PGON`, `PCON` - axis 2;
  - `PARA` -axis 1; axis 2,3 for a special case
- `GSPOSP` is implemented via individual logical volumes for each instantiation.
- `GSROTM` is implemented. Reflections of hierarchies based on plain CSG solids are implemented through the `G3Division` class.
- Hits are not implemented.
- Conversion of GEANT 3.21 magnetic field is currently not supported. However, the usage of magnetic field has to be turned on.

## 4.1.10 Detecting Overlapping Volumes

### The problem of overlapping volumes

Volumes are often positioned within other volumes with the intent that one is fully contained within the other. If, however, a volume extends beyond the boundaries of its mother volume, it is defined as overlapping. It may also be intended that volumes are positioned within the same mother volume such that they do not intersect one another. When such volumes do intersect, they are also defined as overlapping.

The problem of detecting overlaps between volumes is bounded by the complexity of the solid model description. Hence it requires the same mathematical sophistication which is needed to describe the most complex solid topology, in general. However, a tunable accuracy can be obtained by approximating the solids via first and/or second order surfaces and checking their intersections.

In general, the most powerful clash detection algorithms are provided by CAD systems, treating the intersection between the solids in their topological form.

### Detecting overlaps at construction

The GEANT4 geometry modeler provides the ability to detect overlaps of placed volumes (normal placements or parameterised) at the time of construction. This check is optional and can be activated when instantiating a placement (see `G4PVPlacement` constructor in *Placements: single positioned copy*) or a parameterised volume (see `G4PVParameterised` constructor in *Repeated volumes*).

The positioning of that specific volume will be checked against all volumes in the same hierarchy level and its mother volume. Depending on the complexity of the geometry being checked, the check may require considerable CPU time; it is therefore suggested to use it only for debugging the geometry setup and to apply it only to the part of the geometry setup which requires debugging.

The classes `G4PVPlacement` and `G4PVParameterised` also provide a method:

```
G4bool CheckOverlaps(G4int res=1000, G4double tol=0., G4bool verbose=true, G4int maxErr=1)
```

which will force the check for the specified volume, and can be therefore used to verify for overlaps also once the geometry is fully built. The check verifies if each placed or parameterised instance is overlapping with other instances or with its mother volume. A default resolution for the number of points to be generated and verified is provided. The method returns `true` if an overlap occurs. It is also possible to specify a "tolerance" by which overlaps not exceeding such quantity will not be reported and a maximum of overlaps errors for the volume; by default, one overlap per volume is reported.

### Detecting overlaps: built-in kernel commands

Built-in run-time commands to activate verification tests for the user-defined geometry are also provided

```
geometry/test/run
--> to start verification of geometry for overlapping regions
    recursively through the volumes tree.
geometry/test/recursion_start [int]
--> to set the starting depth level in the volumes tree from where
    checking overlaps. Default is level '0' (i.e. the world volume).
    The new settings will then be applied to any recursive test run.
geometry/test/recursion_depth [int]
--> to set the total depth in the volume tree for checking overlaps.
    Default is '-1' (i.e. checking the whole tree).
    Recursion will stop after having reached the specified depth (the
    default being the full depth of the geometry tree).
    The new settings will then be applied to any recursive test run.
geometry/test/tolerance [double] [unit]
--> to define tolerance by which overlaps should not be reported.
    Default is '0'.
geometry/test/verbosity [bool]
--> to set verbosity mode. Default is 'true'.
geometry/test/resolution [int]
--> to establish the number of points on surface to be generated
    and checked for each volume. Default is '10000'.
geometry/test/maximum_errors [int]
--> to fix the threshold for the number of errors to be reported
    for a single volume. By default, for each volume, reports stop
    after the first error reported.
```

To detect overlapping volumes, the built-in UI commands use the random generation of points on surface technique described above. It allows to detect with high level of precision any kind of overlaps, as depicted below. For example, consider Fig. 4.4:

Here we have a line intersecting some physical volume (large, black rectangle). Belonging to the volume are four daughters: A, B, C, and D. Indicated by the dots are the intersections of the line with the mother volume and the four daughters.

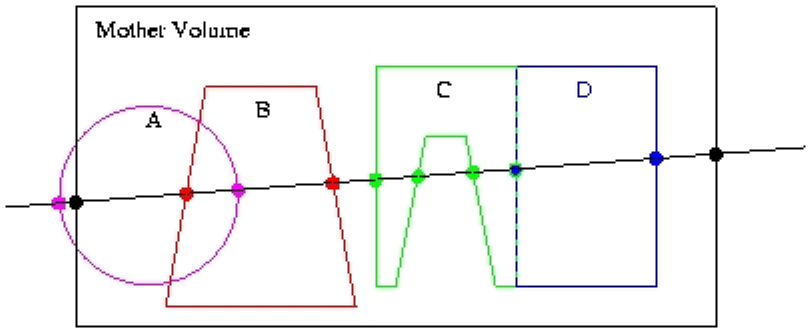


Fig. 4.4: Different cases of placed volumes overlapping each other.

This example has two geometry errors. First, volume A sticks outside its mother volume (this practice, sometimes used in GEANT3.21, is not allowed in GEANT4). This can be noticed because the intersection point (leftmost magenta dot) lies outside the mother volume, as defined by the space between the two black dots.

The second error is that daughter volumes A and B overlap. This is noticeable because one of the intersections with A (rightmost magenta dot) is inside the volume B, as defined as the space between the red dots. Alternatively, one of the intersections with B (leftmost red dot) is inside the volume A, as defined as the space between the magenta dots.

Another difficult issue is roundoff error. For example, daughters C and D lie precisely next to each other. It is possible, due to roundoff, that one of the intersections points will lie just slightly inside the space of the other. In addition, a volume that lies tightly up against the outside of its mother may have an intersection point that just slightly lies outside the mother.

Finally, notice that no mention is made of the possible daughter volumes of A, B, C, and D. To keep the code simple, only the immediate daughters of a volume are checked at one pass. To test these "granddaughter" volumes, the daughters A, B, C, and D each have to be tested themselves in turn. To make this automatic, a recursive algorithm is applied; it first tests the target volume, then it loops over all daughter volumes and calls itself.

**Note:** for a complex geometry, checking the entire volume hierarchy can be extremely time consuming.

### Using built-in visualisation features

See *Debugging geometry with vis*.

### Using the visualization tool DAVID

The GEANT4 visualization offers also a debugging tool for detecting potential intersections of physical volumes. The GEANT4 DAVID visualization tool can automatically detect the overlaps between the volumes defined in GEANT4 and converted to a graphical representation for visualization purposes. The accuracy of the graphical representation can be tuned onto the exact geometrical description. In the debugging, physical-volume surfaces are automatically decomposed into 3D polygons, and intersections of the generated polygons are investigated. If a polygon intersects with another one, physical volumes which these polygons belong to are visualized in color (red is the default). The Fig. 4.5 figure below is a sample visualization of a detector geometry with intersecting physical volumes highlighted:

At present physical volumes made of the following solids can be debugged: `G4Box`, `G4Cons`, `G4Para`, `G4Sphere`, `G4Trd`, `G4Trap`, `G4Tubs`. (Existence of other solids is harmless.)

Visual debugging of physical-volume surfaces is performed with the DAWNFILE driver defined in the visualization category and with the two application packages, i.e. Fukui Renderer "DAWN" and a visual intersection debugger

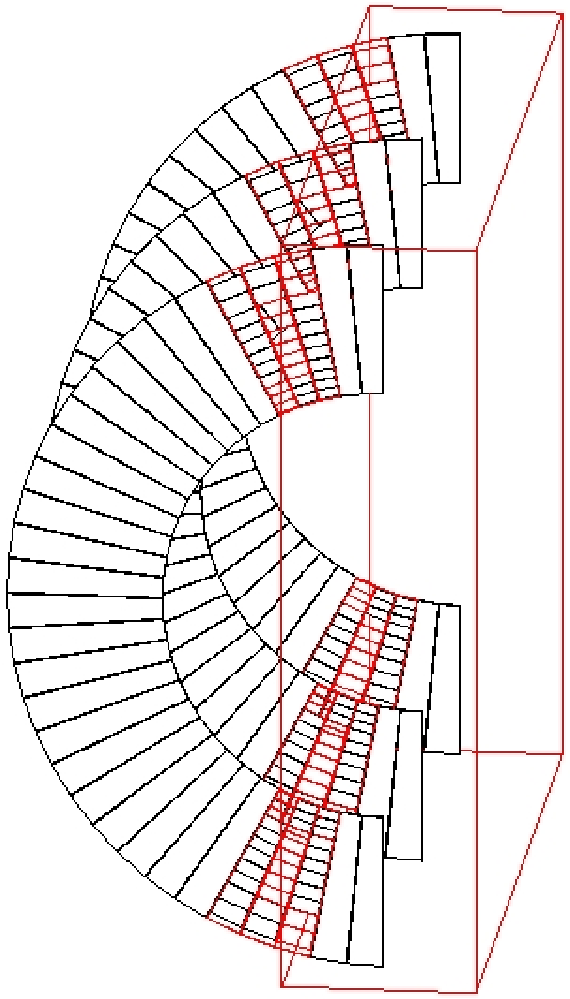


Fig. 4.5: A geometry with overlapping volumes highlighted by DAVID.

"DAVID".

How to compile GEANT4 with the DAWNFILE driver incorporated is described in *The Visualization Drivers*.

If the DAWNFILE driver, DAWN and DAVID are all working well in your host machine, the visual intersection debugging of physical-volume surfaces can be performed as follows:

Run your GEANT4 executable, invoke the DAWNFILE driver, and execute visualization commands to visualize your detector geometry:

```
Idle> /vis/open DAWNFILE
.....(setting camera etc)...
Idle> /vis/drawVolume
Idle> /vis/viewer/update
```

Then a file "g4.prim", which describes the detector geometry, is generated in the current directory and DAVID is invoked to read it. (The description of the format of the file g4.prim can be found from the DAWN web site documentation.)

If DAVID detects intersection of physical-volume surfaces, it automatically invokes DAWN to visualize the detector geometry with the intersected physical volumes highlighted (See the above sample visualization).

If no intersection is detected, visualization is skipped and the following message is displayed on the console:

```
------------------------------------------------------
!!! Number of intersected volumes : 0 !!!
!!! Congratulations ! \(^o^)/         !!!
------------------------------------------------------
```

If you always want to skip visualization, set an environmental variable as follows beforehand:

```
%  setenv DAVID_NO_VIEW  1
```

To control the precision associated to computation of intersections (default precision is set to 9), it is possible to use the environmental variable for the DAWNFILE graphics driver, as follows:

```
%  setenv G4DAWNFILE_PRECISION  10
```

If necessary, re-visualize the detector geometry with intersected parts highlighted. The data are saved in a file "g4david.prim" in the current directory. This file can be re-visualized with DAWN as follows:

```
% dawn g4david.prim
```

It is also helpful to convert the generated file g4david.prim into a VRML-formatted file and perform interactive visualization of it with your WWW browser. The file conversion tool `prim2wrml` can be downloaded from the DAWN web site.

## 4.1.11 Dynamic Geometry Setups

GEANT4 can handle geometries which vary in time (e.g. a geometry varying between two runs in the same job).

It is considered a change to the geometry setup, whenever for the same physical volume:

- the shape or dimension of its related solid is modified;
- the positioning (translation or rotation) of the volume is changed;
- the volume (or a set of volumes, tree) is removed/replaced or added.

Whenever such a change happens, the geometry setup needs to be first "opened" for the change to be applied and afterwards "closed" for the optimisation to be reorganised.

In the general case, in order to notify the GEANT4 system of the change in the geometry setup, the `G4RunManager` has to be messaged once the new geometry setup has been finalised:

```
G4RunManager::GeometryHasBeenModified();
```

The above notification needs to be performed also if a material associated to a *positioned* volume is changed, in order to allow for the internal materials/cuts table to be updated. However, for relatively complex geometries the re-optimisation step may be extremely inefficient, since it has the effect that the whole geometry setup will be re-optimised and re-initialised. In cases where only a limited portion of the geometry has changed, it may be suitable to apply the re-optimisation only to the affected portion of the geometry (subtree).

Since release 7.1 of the GEANT4 toolkit, it is possible to apply re-optimisation local to the subtree of the geometry which has changed. The user will have to explicitly "open/close" the geometry providing a pointer to the top physical volume concerned:

Listing 4.9: Opening and closing a portion of the geometry without notifying the `G4RunManager`.

```
#include "G4GeometryManager.hh"

// Open geometry for the physical volume to be modified ...
//
G4GeometryManager::OpenGeometry(physCalor);

// Modify dimension of the solid ...
//
physCalor->GetLogicalVolume()->GetSolid()->SetXHalfLength(12.5*cm);

// Close geometry for the portion modified ...
//
G4GeometryManager::CloseGeometry(physCalor);
```

If the existing geometry setup is modified locally in more than one place, it may be convenient to apply such a technique only once, by specifying a physical volume on top of the hierarchy (subtree) containing all changed portions of the setup.

An alternative solution for dealing with dynamic geometries is to specify NOT to apply optimisation for the subtree affected by the change and apply the general solution of invoking the `G4RunManager`. In this case, a performance penalty at run-time may be observed (depending on the complexity of the not-optimised subtree), considering that, without optimisation, intersections to all volumes in the subtree will be explicitly computed each time.

**Note:** in multi-threaded runs, dynamic geometries are only allowed for runs consisting only of one event.

### 4.1.12 Importing XML Models Using GDML

Geometry Description Markup Language (GDML) is a markup language based on XML and suited for the description of detector geometry models. It allows for easy exchange of geometry data in a *human-readable* XML-based description and structured formatting.

The GDML parser is a component of GEANT4 which can be built and installed as an optional choice. It allows for importing and exporting GDML files, following the schema specified in the GDML documentation. The installation of the plugin is optional and requires the installation of the XercesC DOM parser.

Examples of how to import and export a detector description model based on GDML, and also how to extend the GDML schema, are provided and can be found in `examples/extended/persistency/gdml`.

### 4.1.13 Importing ASCII Text Models

Since release 9.2 of GEANT4, it is also possible to import geometry setups based on a plain text description, according to a well defined syntax for identifying the different geometrical entities (solids, volumes, materials and volume attributes) with associated parameters. An example showing how to define a geometry in plain text format and import it in a GEANT4 application is shown in `examples/extended/persistency/P03`. The example also covers the case of associating a sensitive detector to one of the volumes defined in the text geometry, the case of mixing C++ and text geometry definitions and the case of defining new tags in the text format so that regions and cuts by region can be defined in the text file. It also provides an example of how to write a geometry text file from the in-memory GEANT4 geometry. For the details on the format see the dedicated manual.

### 4.1.14 Saving geometry tree objects in binary format

The GEANT4 geometry tree can be stored in the Root binary file format using the *Root-I/O* technique provided by in Root. Such a binary file can then be used to quickly load the geometry into the memory or to move geometries between different GEANT4 applications.

See *Object Persistency* for details and references.

## 4.2 Material

### 4.2.1 General considerations

In nature, materials (chemical compounds, mixtures) are made of elements, and elements are made of isotopes. GEANT4 has three main classes designed to reflect this organization. Each of these classes has a table, which is a static data member, used to keep track of the instances of the respective classes created.

**`G4Isotope`** This class describes the properties of atoms: atomic number, number of nucleons, mass per mole, etc.

**`G4Element`** This class describes the properties of elements: effective atomic number, effective number of nucleons, effective mass per mole, number of isotopes, shell energy, and quantities like cross section per atom, etc.

**`G4Material`** This class describes the macroscopic properties of matter: density, state, temperature, pressure, and macroscopic quantities like radiation length, mean free path, dE/dx, etc.

Only the `G4Material` class is visible to the rest of the toolkit and used by the tracking, the geometry and the physics. It contains all the information relevant to its constituent elements and isotopes, while at the same time hiding their implementation details.

### 4.2.2 Introduction to the Classes

#### G4Isotope

A `G4Isotope` object has a name, atomic number, number of nucleons, mass per mole, and an index in the table. The constructor automatically stores “this” isotope in the isotopes table, which will assign it an index number. The `G4Isotope` objects are owned by the isotopes table, and must not be deleted by user code.

### G4Element

A `G4Element` object has a name, symbol, effective atomic number, effective number of nucleons, effective mass of a mole, an index in the elements table, the number of isotopes, a vector of pointers to such isotopes, and a vector of relative abundances referring to such isotopes (where relative abundance means the number of atoms per volume). In addition, the class has methods to add, one by one, the isotopes which are to form the element.

The constructor automatically stores "this" element in the elements table, which will assign it an index number. The `G4Element` objects are owned by the elements table, and must not be deleted by user code.

A `G4Element` object can be constructed by directly providing the effective atomic number, effective number of nucleons, and effective mass of a mole, if the user explicitly wants to do so. Alternatively, a `G4Element` object can be constructed by declaring the number of isotopes of which it will be composed. The constructor will "new" a vector of pointers to `G4Isotopes` and a vector of doubles to store their relative abundances. Finally, the method to add an isotope must be invoked for each of the desired (pre-existing) isotope objects, providing their addresses and relative abundances. At the last isotope entry, the system will automatically compute the effective atomic number, effective number of nucleons and effective mass of a mole, and will store "this" element in the elements table.

A few quantities, with physical meaning or not, which are constant in a given element, are computed and stored here as "derived data members".

Using the internal GEANT4 database, a `G4Element` can be accessed by atomic number or by atomic symbol ("Al", "Fe", "Pb"…). In that case `G4Element` will be found from the list of existing elements or will be constructed using data from the GEANT4 database, which is derived from the NIST database of elements and isotope compositions. Thus, the natural isotope composition can be built by default. The same element can be created as using the NIST database with the natural composition of isotopes and from scratch in user code with user defined isotope composition.

### G4Material

A `G4Material` object has a name, density, physical state, temperature and pressure (by default the standard conditions), the number of elements and a vector of pointers to such elements, a vector of the fraction of mass for each element, a vector of the atoms (or molecules) numbers of each element, and an index in the materials table. In addition, the class has methods to add, one by one, the elements which will comprise the material.

The constructor automatically stores "this" material in the materials table, which will assign it an index number. The `G4Material` objects are owned by the materials table, and must not be deleted by user code.

A `G4Material` object can be constructed by directly providing the resulting effective numbers, if the user explicitly wants to do so (an underlying element will be created with these numbers). Alternatively, a `G4Material` object can be constructed by declaring the number of elements of which it will be composed. The constructor will "new" a vector of pointers to `G4Element` and a vector of doubles to store their fraction of mass. Finally, the method to add an element must be invoked for each of the desired (pre-existing) element objects, providing their addresses and mass fractions. At the last element entry, the system will automatically compute the vector of the number of atoms of each element per volume, the total number of electrons per volume, and will store "this" material in the materials table. In the same way, a material can be constructed as a mixture of other materials and elements.

It should be noted that if the user provides the number of atoms (or molecules) for each element comprising the chemical compound, the system automatically computes the mass fraction. A few quantities, with physical meaning or not, which are constant in a given material, are computed and stored here as "derived data members".

Some materials are included in the internal GEANT4 database, which were derived from the NIST database of material properties. Additionally a number of materials frequently used in HEP is included in the database. Materials are interrogated or constructed by their *names* ( *Material Database*). There are UI commands for the material category, which provide an interactive access to the database. If material is created using the NIST database by it will consist by default of elements with the natural composition of isotopes.

### Final Considerations

The classes will automatically decide if the total of the mass fractions is correct, and perform the necessary checks. The main reason why a fixed index is kept as a data member is that many cross section and energy tables will be built in the physics processes “by rows of materials (or elements, or even isotopes)”. The tracking gives the physics process the address of a material object (the material of the current volume). If the material has an index according to which the cross section table has been built, then direct access is available when a number in such a table must be accessed. We get directly to the correct row, and the energy of the particle will tell us the column. Without such an index, every access to the cross section or energy tables would imply a search to get to the correct material’s row. More details will be given in the section on processes.

Isotopes, elements and materials must be instantiated dynamically in the user application; they are automatically registered in internal stores and the system takes care to free the memory allocated at the end of the job.

## 4.2.3 Recipes for Building Elements and Materials

The Listing 4.10 illustrates the different ways to define materials.

Listing 4.10: A program which illustrates the different ways to define materials.

```
#include "G4Isotope.hh"
#include "G4Element.hh"
#include "G4Material.hh"
#include "G4UnitsTable.hh"

int main() {
G4String name, symbol;             // a=mass of a mole;
G4double a, z, density;            // z=mean number of protons;
G4int iz, n;                       // iz=nb of protons  in an isotope;
                                   // n=nb of nucleons in an isotope;
G4int ncomponents, natoms;
G4double abundance, fractionmass;
G4double temperature, pressure;

G4UnitDefinition::BuildUnitsTable();

// define Elements
a = 1.01*g/mole;
G4Element* elH  = new G4Element(name="Hydrogen",symbol="H" , z= 1., a);

a = 12.01*g/mole;
G4Element* elC  = new G4Element(name="Carbon"  ,symbol="C" , z= 6., a);

a = 14.01*g/mole;
G4Element* elN  = new G4Element(name="Nitrogen",symbol="N" , z= 7., a);

a = 16.00*g/mole;
G4Element* elO  = new G4Element(name="Oxygen"  ,symbol="O" , z= 8., a);

a = 28.09*g/mole;
G4Element* elSi = new G4Element(name="Silicon", symbol="Si", z=14., a);

a = 55.85*g/mole;
G4Element* elFe = new G4Element(name="Iron"    ,symbol="Fe", z=26., a);

a = 183.84*g/mole;
G4Element* elW = new G4Element(name="Tungsten" ,symbol="W",  z=74., a);

a = 207.20*g/mole;
G4Element* elPb = new G4Element(name="Lead"    ,symbol="Pb", z=82., a);
```

(continues on next page)

(continued from previous page)

```
// define an Element from isotopes, by relative abundance
G4Isotope* U5 = new G4Isotope(name="U235", iz=92, n=235, a=235.01*g/mole);
G4Isotope* U8 = new G4Isotope(name="U238", iz=92, n=238, a=238.03*g/mole);

G4Element* elU  = new G4Element(name="enriched Uranium", symbol="U", ncomponents=2);
elU->AddIsotope(U5, abundance= 90.*perCent);
elU->AddIsotope(U8, abundance= 10.*perCent);

G4cout << *(G4Isotope::GetIsotopeTable()) << G4endl;
G4cout << *(G4Element::GetElementTable()) << G4endl;
```

```
// define simple materials
density = 2.700*g/cm3;
a = 26.98*g/mole;
G4Material* Al = new G4Material(name="Aluminum", z=13., a, density);

density = 1.390*g/cm3;
a = 39.95*g/mole;
G4Material* lAr = new G4Material(name="liquidArgon", z=18., a, density);

density = 8.960*g/cm3;
a = 63.55*g/mole;
G4Material* Cu = new G4Material(name="Copper"   , z=29., a, density);

// define a material from elements.   case 1: chemical molecule
density = 1.000*g/cm3;
G4Material* H2O = new G4Material(name="Water", density, ncomponents=2);
H2O->AddElement(elH, natoms=2);
H2O->AddElement(elO, natoms=1);

density = 1.032*g/cm3;
G4Material* Sci = new G4Material(name="Scintillator", density, ncomponents=2);
Sci->AddElement(elC, natoms=9);
Sci->AddElement(elH, natoms=10);

density = 2.200*g/cm3;
G4Material* SiO2 = new G4Material(name="quartz", density, ncomponents=2);
SiO2->AddElement(elSi, natoms=1);
SiO2->AddElement(elO , natoms=2);

density = 8.280*g/cm3;
G4Material* PbWO4= new G4Material(name="PbWO4", density, ncomponents=3);
PbWO4->AddElement(elO , natoms=4);
PbWO4->AddElement(elW , natoms=1);
PbWO4->AddElement(elPb, natoms=1);

// define a material from elements.   case 2: mixture by fractional mass
density = 1.290*mg/cm3;
G4Material* Air = new G4Material(name="Air  "  , density, ncomponents=2);
Air->AddElement(elN, fractionmass=0.7);
Air->AddElement(elO, fractionmass=0.3);

// define a material from elements and/or others materials (mixture of mixtures)
density = 0.200*g/cm3;
G4Material* Aerog = new G4Material(name="Aerogel", density, ncomponents=3);
Aerog->AddMaterial(SiO2, fractionmass=62.5*perCent);
Aerog->AddMaterial(H2O , fractionmass=37.4*perCent);
Aerog->AddElement (elC , fractionmass= 0.1*perCent);
```

```
// examples of gas in non STP conditions
density     = 27.*mg/cm3;
pressure    = 50.*atmosphere;
temperature = 325.*kelvin;
G4Material* CO2 = new G4Material(name="Carbonic gas", density, ncomponents=2,
```

(continues on next page)

(continued from previous page)

```
                                    kStateGas,temperature,pressure);
CO2->AddElement(elC, natoms=1);
CO2->AddElement(elO, natoms=2);

density     = 0.3*mg/cm3;
pressure    = 2.*atmosphere;
temperature = 500.*kelvin;
G4Material* steam = new G4Material(name="Water steam ", density, ncomponents=1,
                                     kStateGas,temperature,pressure);
steam->AddMaterial(H2O, fractionmass=1.);

// What about vacuum ?  Vacuum is an ordinary gas with very low density
density     = universe_mean_density;                //from PhysicalConstants.h
pressure    = 1.e-19*pascal;
temperature = 0.1*kelvin;
new G4Material(name="Galactic", z=1., a=1.01*g/mole, density,
                   kStateGas,temperature,pressure);

density     = 1.e-5*g/cm3;
pressure    = 2.e-2*bar;
temperature = STP_Temperature;                      //from PhysicalConstants.h
G4Material* beam = new G4Material(name="Beam ", density, ncomponents=1,
                                     kStateGas,temperature,pressure);
beam->AddMaterial(Air, fractionmass=1.);

// print the table of materials
G4cout << *(G4Material::GetMaterialTable()) << G4endl;

return EXIT_SUCCESS;
}
```

As can be seen in the later examples, a material has a state: solid (the default), liquid, or gas. The constructor checks the density and automatically sets the state to gas below a given threshold (10 mg/cm3).

In the case of a gas, one may specify the temperature and pressure. The defaults are STP conditions defined in `PhysicalConstants.hh`.

An element must have the number of nucleons >= number of protons >= 1.

A material must have non-zero values of density, temperature and pressure.

Materials can also be defined using the internal GEANT4 database. Listing 4.11 illustrates how to do this for the same materials used in Listing 4.10. There are also UI commands which allow the database to be accessed. *The list of currently available material names* ( *Material Database*) is extended permanently.

Listing 4.11: A program which shows how to define materials from the internal database.

```
#include "globals.hh"
#include "G4Material.hh"
#include "G4NistManager.hh"

int main() {
  G4NistManager* man = G4NistManager::Instance();
  man->SetVerbose(1);

  // define elements
  G4Element* C  = man->FindOrBuildElement("C");
  G4Element* Pb = man->FindOrBuildMaterial("Pb");

  // define pure NIST materials
  G4Material* Al = man->FindOrBuildMaterial("G4_Al");
  G4Material* Cu = man->FindOrBuildMaterial("G4_Cu");
```

(continues on next page)

(continued from previous page)

```
  // define NIST materials
  G4Material* H2O  = man->FindOrBuildMaterial("G4_WATER");
  G4Material* Sci  = man->FindOrBuildMaterial("G4_PLASTIC_SC_VINYLTOLUENE");
  G4Material* SiO2 = man->FindOrBuildMaterial("G4_SILICON_DIOXIDE");
  G4Material* Air  = man->FindOrBuildMaterial("G4_AIR");

  // HEP materials
  G4Material* PbWO4  = man->FindOrBuildMaterial("G4_PbWO4");
  G4Material* lAr    = man->FindOrBuildMaterial("G4_lAr");
  G4Material* vac    = man->FindOrBuildMaterial("G4_Galactic");

  // define gas material at non STP conditions (T = 120K, P=0.5atm)
  G4Material* coldAr = man->ConstructNewGasdMaterial("ColdAr","G4_Ar",120.*kelvin,0.
↪5*atmosphere);

  // print the table of materials
  G4cout << *(G4Material::GetMaterialTable()) << G4endl;

  return EXIT_SUCCESS;
}
```

### 4.2.4 The Tables

#### Print a constituent

The following shows how to print a constituent:

```
G4cout << elU << G4endl;
G4cout << Air << G4endl;
```

#### Print the table of materials

The following shows how to print the table of materials:

```
G4cout << *(G4Material::GetMaterialTable()) << G4endl;
```

## 4.3 Electromagnetic Field

### 4.3.1 An Overview of Propagation in a Field

GEANT4 is capable of describing and propagating in a variety of fields. Magnetic fields, electric fields, electromagnetic fields, and gravity fields, uniform or non-uniform, can specified for a GEANT4 setup. The propagation of tracks inside them can be performed to a user-defined accuracy.

In order to propagate a track inside a field, the equation of motion of the particle in the field is integrated. In general, this is done using a Runge-Kutta method for the integration of ordinary differential equations. However, for specific cases where an analytical solution is known, it is possible to utilize this instead. Several Runge-Kutta methods are available, suitable for different conditions. In specific cases (such as a uniform field where the analytical solution is known) different solvers can also be used. In addition, when an approximate analytical solution is known, it is possible to utilize it in an iterative manner in order to converge to the solution to the precision required. This latter method is currently implemented and can be used particularly well for magnetic fields that are almost uniform.

Once a method is chosen that calculates the track's propagation in a specific field, the curved path is broken up into linear chord segments. These chord segments are determined so that they closely approximate the curved path. The chords are then used to interrogate the Navigator as to whether the track has crossed a volume boundary. Several parameters are available to adjust the accuracy of the integration and the subsequent interrogation of the model geometry.

How closely the set of chords approximates a curved trajectory is governed by a parameter called the *miss distance* (also called the *chord distance*). This is an upper bound for the value of the sagitta - the distance between the 'real' curved trajectory and the approximate linear trajectory of the chord. By setting this parameter, the user can control the precision of the volume interrogation. Every attempt has been made to ensure that all volume interrogations will be made to an accuracy within this *miss distance*.

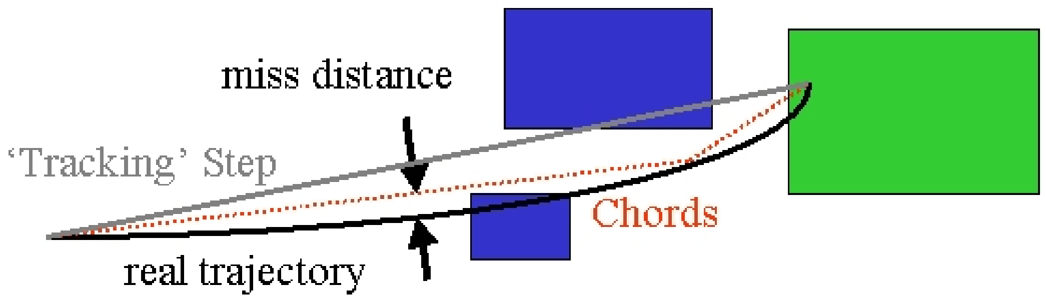


Fig. 4.6: The curved trajectory will be approximated by chords, so that the maximum estimated distance and chord is less than the *miss distance*.

In addition to the *miss distance* there are two more parameters which the user can set in order to adjust the accuracy (and performance) of tracking in a field. In particular these parameters govern the accuracy of the intersection with a volume boundary and the accuracy of the integration of other steps. As such they play an important role for tracking.

The *delta intersection* parameter is the accuracy to which an intersection with a volume boundary is calculated. If a candidate boundary intersection is estimated to have a precision better than this, it is accepted. This parameter is especially important because it is used to limit a bias that our algorithm (for boundary crossing in a field) exhibits. This algorithm calculates the intersection with a volume boundary using a chord between two points on the curved particle trajectory. As such, the intersection point is always on the 'inside' of the curve. By setting a value for this parameter that is much smaller than some acceptable error, the user can limit the effect of this bias on, for example, the future estimation of the reconstructed particle momentum.

The *delta one step* parameter is the accuracy for the endpoint of 'ordinary' integration steps, those which do not intersect a volume boundary. This parameter is a limit on the estimated error of the endpoint of each physics step. It can be seen as akin to a statistical uncertainty and is not expected to contribute any systematic behavior to physical quantities. In contrast, the bias addressed by *delta intersection* is clearly correlated with potential systematic errors in the momentum of reconstructed tracks. Thus very strict limits on the intersection parameter should be used in tracking detectors or wherever the intersections are used to reconstruct a track's momentum.

*Delta intersection* and *delta one step* are parameters of the Field Manager; the user can set them according to the demands of his application. Because it is possible to use more than one field manager, different values can be set for different detector regions.

Note that reasonable values for the two parameters are strongly coupled: it does not make sense to request an accuracy of 1 nm for *delta intersection* and accept 100 $\mu$m for the *delta one step* error value. Nevertheless *delta intersection* is the more important of the two. It is recommended that these parameters should not differ significantly - certainly not by more than an order of magnitude.

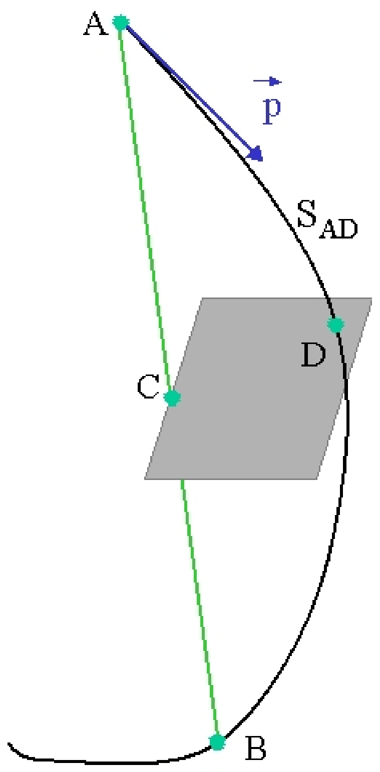


Fig. 4.7: The distance between the calculated chord intersection point C and a computed curve point D is used to determine whether C is an accurate representation of the intersection of the curved path ADB with a volume boundary. Here CD is likely too large, and a new intersection on the chord AD will be calculated.

### 4.3.2 Practical Aspects

#### Creating a Magnetic Field for a Detector

The simplest way to define a field for a detector involves the following steps:

1. create a field. It can be uniform

   ```
   #include "G4SystemOfUnits.hh"

   G4MagneticField *magField;
   magField = new G4UniformMagField(G4ThreeVector(0.,0.,3.0*kilogauss));
   ```

   or non-uniform:

   ```
   magField = new G4QuadrupoleMagField( 1.*tesla/(1.*meter) );
   ```

2. set it as the default field:

   ```
   G4FieldManager* fieldMgr
     = G4TransportationManager::GetTransportationManager()
       ->GetFieldManager();
   fieldMgr->SetDetectorField(magField);
   ```

3. create the objects which calculate the trajectory for a pure magnetic field,:

   ```
   fieldMgr->CreateChordFinder(magField);
   ```

   This is a short cut, which creates all the equation of motion, a method for integration (Runge-Kutta stepper) and the driver which controls the integration and limits its estimated error.

#### Creating a Uniform Magnetic Field with user commands

Since 10.0 version, it is also possible to create a uniform magnetic field and perform the other two steps above the `G4GlobalMagFieldMessenger` class:

```
G4ThreeVector fieldValue = G4ThreeVector(0.,0.,fieldValue);
fMagFieldMessenger = new G4GlobalMagFieldMessenger(fieldValue);
fMagFieldMessenger->SetVerboseLevel(1);
```

The messenger creates the global uniform magnetic field, which is activated (set to the `G4TransportationManager` object) only when the `fieldValue` is non zero vector. The messenger class setter functions can be then used to change the field value (and activate or inactivate the field again) or the level of output messages. The messenger also takes care of deleting the field.

As its class name suggests, the messenger creates also UI commands which can be used to change the field value and the verbose level interactively or from a macro:

```
/globalField/setValue vx vy vz unit
/globalField/verbose level
```

### Creating a Field for a Part of the Volume Hierarchy

It is possible to create a field for a part of the detector. In particular it can describe the field (with pointer pEmField, for example) inside a logical volume and all its daughters. This can be done by simply creating a `G4FieldManager` and attaching it to a logical volume (with pointer, logicVolumeWithField, for example) or set of logical volumes.

```
G4bool allLocal = true;
logicVolumeWithField->SetFieldManager(localFieldManager, allLocal);
```

Using the second parameter to `SetFieldManager` you choose whether daughter volumes of this logical volume will also be given this new field. If it has the value `true`, the field will be assigned also to its daughters, and all their sub-volumes. Else, if it is `false`, it will be copied only to those daughter volumes which do not have a field manager already, and recursively to their sub-volumes without a field manager.

### Creating an Electric or Electromagnetic Field

The design and implementation of the *Field* category allows and enables the use of an electric or combined electromagnetic field. These fields can also vary with time, as can magnetic fields.

Source listing Listing 4.12 shows how to define a uniform electric field for the whole of a detector.

Listing 4.12: How to define a uniform electric field for the whole of a detector, extracted from example in examples/extended/field/field02 .

```
// in the header file (or first)
#include "G4EqMagElectricField.hh"
#include "G4UniformElectricField.hh"
#include "G4DormandPrince745.hh"
...
G4ElectricField*        pEMfield;
G4EqMagElectricField*   pEquation;
G4ChordFinder*          pChordFinder ;

// in the source file

{
  pEMfield = new G4UniformElectricField(
               G4ThreeVector(0.0,100000.0*kilovolt/cm,0.0));

  // Create an equation of motion for this field
  pEquation = new G4EqMagElectricField(pEMfield);

  G4int nvar = 8;

  // Create the Runge-Kutta 'stepper' using the efficient 'DoPri5' method
  auto pStepper = new G4DormandPrince745( pEquation, nvar );

  // Get the global field manager
  auto fieldManager= G4TransportationManager::GetTransportationManager()->
       GetFieldManager();
  // Set this field to the global field manager
  fieldManager->SetDetectorField( pEMfield );

  G4double minStep     = 0.010*mm ; // minimal step of 10 microns

  // The driver will ensure that integration is control to give
  //   acceptable integration error
  auto pIntgrationDriver =
     new G4IntegrationDriver<G4DormandPrince745>(minStep,
                                                 pStepper,
                                                 nvar);
```

(continues on next page)

(continued from previous page)

```
  pChordFinder = new G4ChordFinder(pIntgrationDriver);
  fieldManager->SetChordFinder( pChordFinder );
}
```

An example with an electric field is examples/extended/field/field02, where the class F02ElectricFieldSetup demonstrates how to set these and other parameters, and how to choose different Integration Steppers. An example with a uniform gravity field (G4UniformGravityField) is examples/extended/field/field06.

Note that using gravity since Geant4 10.6 it is necessary to enable it in the transportation process(es) used in the simulation. ( This is in order to enable optimisations which are possible only in its absence. )

The user can also create their own type of field, inheriting from `G4VField`, and an associated Equation of Motion class (inheriting from `G4EqRhs`) to simulate other types of fields.

### How to Adjust the Accuracy of hitting a volume

Straight-line chord segments are used to detect volume boundary crossing. The curved trajectory is broken up into such segments using an accuracy parameter `DeltaChord`. Segments much be chosene so that their 'sagitta', the maximum distance between the curve and chord, is smaller than `DeltaChord`. So effectively this is the maximum distance by which a volume that should be intersected could be missed.

To change the accuracy of the approximation of the curved trajectory by linear segments, use the `SetDeltaChord` method:

```
fieldMgr->GetChordFinder()->SetDeltaChord( dcLength ); // Units: length
```

Geant4 propagation will seek ensure that any volume within `dcLenght` from the curved trajectory will be intersected.

### How to Adjust the Integration Accuracy

In order to obtain a particular accuracy in tracking particles through an electromagnetic field, it is necessary to adjust the parameters of the field propagation module. In the following section, some of these additional parameters are discussed.

When integration is used to calculate the trajectory, it is necessary to determine an acceptable level of numerical imprecision in order to get performant simulation with acceptable errors. The parameters in GEANT4 tell the field module what level of integration inaccuracy is acceptable.

In all quantities which are integrated (position, momentum, energy) there will be errors. Here, however, we focus on the error in two key quantities: the position and the momentum. (The error in the energy will come from the momentum integration).

Three parameters exist which are relevant to the integration accuracy. DeltaOneStep is a distance and is roughly the position error which is acceptable in an integration step. Since many integration steps may be required for a single physics step, DeltaOneStep should be a fraction of the average physics step size. The next two parameters impose a further limit on the relative error of the position/momentum inaccuracy. EpsilonMin and EpsilonMax impose a minimum and maximum on this relative error - and take precedence over DeltaOneStep. (Note: if you set EpsilonMin=EpsilonMax=your-value, then all steps will be made to this relative precision.

Listing 4.13: How to set accuracy parameters for the 'global' field of the setup.

```
G4FieldManager *globalFieldManager;

G4TransportationManager *transportMgr=
     G4TransportationManager::GetTransportationManager();
```

(continues on next page)

(continued from previous page)

```
globalFieldManager = transportMgr->GetFieldManager();
                          // Relative accuracy values:
G4double minEps= 1.0e-5;  //   Minimum & value for largest steps
G4double maxEps= 1.0e-4;  //   Maximum & value for smallest steps

globalFieldManager->SetMinimumEpsilonStep( minEps );
globalFieldManager->SetMaximumEpsilonStep( maxEps );
globalFieldManager->SetDeltaOneStep( 0.5e-3 * mm );  // 0.5 micrometer

G4cout << "EpsilonStep: set min= " << minEps << " max= " << maxEps << G4endl;
```

We note that the relevant parameters above limit the inaccuracy in each step. The final inaccuracy due to the full trajectory will accumulate!

The exact point at which a track crosses a boundary is also calculated with finite accuracy. To limit this inaccuracy, a parameter called DeltaIntersection is used. This is a maximum for the inaccuracy of a single boundary crossing. Thus the accuracy of the position of the track after a number of boundary crossings is directly proportional to the number of boundaries.

### Full control of integration method for a magnetic field

You can instead specify explicitly the full set of classes for propagating in a magnetic field. This provides full control over the method of integration, and allows the choice of higher or lower order methods. It also all you to select the use of methods which used to be the default choice in the past (e.g. `G4ClassicalRungeRK4` or `G4DormandPrince745` without using interpolation.)

The classes required are the equation of motion:

```
auto pEquation = new G4Mag_UsualEqRhs(magField);
G4int nVar= pEquation->GetNumberOfVariables();
```

the method of integration (stepper):

```
auto pStepper =  new G4DormandPrince745( pEquation );
```

the driver to control the accuracy of integration:

```
auto driver = G4InterpolationDriver<G4DormandPrince745>(minStep,pStepper, nvar);
```

or alternatively a driver without interpolation:

```
auto driver= G4IntegrationDriver<typeof(pStepper)>(minStep,pStepper, nvar);
```

and the chord finder:

```
auto chordFinder = new G4ChordFinder( driver );
```

### Choosing a Stepper

Runge-Kutta integration is used to compute the motion of a charged track in a general field. There are many general steppers from which to choose, of low and high order, and specialized steppers for pure magnetic fields. By default, GEANT4 uses the established stepper of Dormand and Prince Runge-Kutta stepper, which is general purpose, efficient and robust. It is a 5th order method which provides an error estimate directly , and requires fewer evaluations of the derivative (and field) than the previous default, the classical 4th order method (for which an error estimate required multiple sub-steps).

For somewhat smooth fields, which change smoothly over the length scales of typical physics steps, there is choice between fifth order steppers (such as the default `G4DormandPrince745`):

```
G4int nvar = 8;  // To integrate time & energy
                 //    in addition to position, momentum
G4EqMagElectricField* pEquation= new G4EqMagElectricField(pEMfield);

auto doPri5stepper = new G4DormandPrince745( pEquation, nvar );
  //  The recommended stepper, well suited for reasonably smooth fields
  //   and intermediate accuracy requirements ( 10^-4 to 10^-7 )
```

Alternative fifth order embedded steppers beside the recommended and default `G4DormandPrince745` which requires 7 field evaluations (stages) include the older `G4CashKarpRKF45` which requires fewer field evaluations (6 ‘stages’)

```
auto CK45stepper = new G4CashKarpRKF45( pEquation, nvar );
   // Alternative 4/5th order stepper for reasonably smooth fields
```

The newest experimental classes `G4BogackiShampine45` or `G4TsitourasRK45` implement some of the most efficient fifth order methods in the literature, but require an additional derivative (field evaluation) per step.:

```
auto BS45stepper = new G4BogackiShampine45( pEquation, nvar );
   // Alternative 4/5th order stepper with 8 stages (evaluations).
```

If there are particularly challenging accuracy demands (better than 1e-7) it may be worth to investigate higher order steppers. Alternatively, if the field is known to have specific properties, lower or higher order steppers can be used to obtain the results of the necessary accuracy using fewer computing cycles.

Since Geant4 10.5 it is recommended to use the templated driver G4IntegrationDriver together with the stepper:

```
auto dp45driver =
  new G4IntegrationDriver<G4DormandPrince745>(stepMin, doPri5stepper, nvar);
```

### Steppers for rough fields

Sometimes the field changes greatly over short distances, and is estimated in ways that do not ensure that its derivatives are smooth. These can present a challenge for fourth or fifth order Runge-Kutta methods.

What matters is the variation of the field in geometrical regions in which a large fraction of particles are tracked. In particular, if the field is calculated from a field map and it varies significantly and in a non-smooth way over short distances in important regions, it is suggested to investigate a lower order stepper.

Steppers of reduced order are also suitable when lower accuracy is required, such as errors of order $10^{-3}$. Such accuracy could be suitable for the least important tracks, such as low energy electrons near the end of their trajectory (but still inside material.)

Steppers of reduced order require fewer derivative evaluations per step. The choice of lower order steppers includes the third order embedded stepper `G4BogackiShampine23`, which provides a direct error estimate.:

```
auto stepper = new G4BogackiShampine23( pEquation, nvar );
  //  3rd order embbedded stepper
  //  Suitable for lower accuracy needs (<~ 10^-3) and/or 'rough' fields
```

Older type steppers, which do not provide a direct error estimate, offer an alternative for the roughest fields. ( Note: these methods estimate the error in a step by subdividing it into two smaller steps and using the difference between the new estimate and the estimate for the whole step as the estimated error. )

The recommended ones are the fourth order `G4ClassicalRK4`, which was the default in releases of GEANT4 up to 10.3, and is very robust:

```
pStepper = new ClassicalRK4( pEquation, nvar );
    //  4th  order, the old default - a robust alternative
```

the third order stepper `G4SimpleHeum`, and the second order steppers `G4ImplicitEuler` and `G4SimpleRunge`.:

```
pStepper = new G4SimpleHeum( pEquation, nvar );
    //  3rd order robust alternative for low accuracy and/or rought fields

pStepper = new G4SimpleRunge( pEquation, nvar );
    //  2nd  order, for very rough (non-smooth) fields
```

A first order stepper is not recommended, but may be used only for the roughest fields, as a cross check for other higher performance methods.

For somewhat smooth fields (intermediate), the choice between a fifth order stepper (such as the default `G4DormandPrince745`):

```
pStepper = new G4DormandPrince745( pEquation, nvar );
  //  The recommended stepper, well suited for reasonably smooth fields
```

embedded third, the older type second or third order steppers, or the established fourth order `G4ClassicalRK4` or

should be made by trial and error.

Trying a few different types of steppers for a particular field or application is suggested if maximum performance is a goal.

The choice of stepper depends on the type of field: magnetic or general. A general field can be an electric or electromagnetic field, it can be a magnetic field or a user-defined field (which requires a user-defined equation of motion.)

For a general field all the above steppers are potential alternatives to the recommended / default `G4DormandPrince745`.

But specialized steppers for pure magnetic fields are also available. The `G4NystromRK4` stepper is a fourth order method which estimates the integration error in a step directly from the variation of the field at the initial point, the midpoint and near the endpoint of the step. Thus it requires no additional evaluations (stages.):

```
G4Mag_UsualEqRhs*
   pEquation = new G4Mag_UsualEqRhs(fMagneticField);
pStepper = new G4NystromRK4( pEquation );
```

Others take into account the fact that a local trajectory in a slowly varying field will not vary significantly from a helix. Combining this in with a variation the Runge-Kutta method can provide higher accuracy at lower computational cost when large steps are possible.

```
pStepper = new G4HelixImplicitEuler( pEquation );
// Note that for magnetic field that do not vary with time,
//  the default number of variables suffices.

// or ..
pStepper = new G4HelixExplicitEuler( pEquation );
pStepper = new G4HelixSimpleRunge( pEquation );
```

A new stepper for propagation in magnetic field is available in release 9.3. Choosing the G4NystromRK4 stepper provides accuracy near that of G4ClassicalRK4 (4th order) with a significantly reduced cost in field evaluation. Using a novel analytical expression for estimating the error of a proposed step and the Nystrom reuse of the mid-point field value, it requires only 2 additional field evaluations per attempted step, in place of 10 field evaluations of ClassicalRK4 (which uses the general midpoint method for estimating the step error.)

```
G4Mag_UsualEqRhs*
   pMagFldEquation = new G4Mag_UsualEqRhs(fMagneticField);
pStepper = new G4NystromRK4( pMagFldEquation );
```

It is proposed as an alternative stepper in the case of a pure magnetic field. It is not applicable for the simulation of electric or full electromagnetic or other types of field. For a pure magnetic field, results should be fully compatible with the results of ClassicalRK4 in nearly all cases. (The only potential exceptions are large steps for tracks with small momenta - which cannot be integrated well by any RK method except the Helical extended methods.)

You can choose an alternative stepper either when the field manager is constructed or later. At the construction of the ChordFinder it is an optional argument:

```
G4ChordFinder( G4MagneticField* itsMagField,
               G4double         stepMinimum = 1.0e-2 * mm,
               G4MagIntegratorStepper* pItsStepper = 0 );
```

To change the stepper at a later time use

```
pChordFinder->GetIntegrationDriver()
            ->RenewStepperAndAdjust( newStepper );
```

### Increasing efficiency with interpolation and FSAL stepper

Often a significant fraction of CPU time is spent in integrating the motion of charged particles in field. This is particularly the case when the cost of evaluating the field at a location (and possibly time) is significant. To improve on this developments over the past years have introduced methods that require fewer field evaluations for the same overall accuracy.

New in GEANT4 10.6 is the ability to full use of the newest RK methods, which have an interpolation capability. Such Runge-Kutta methods provide an interpolation polynomial which can be evaluated to estimate the values of all integrated variables at an arbitrary intermediate length in the integration interval.

Both these interpolation capabilities are harnessed by the new type of integration driver `G4InterpolationDriver`. Currently this combination is available only with the `G4DormandPrince745` stepper.

```
using InterpolationDriverType = G4InterpolationDriver<G4DormandPrince745>;
auto dopri5stepper = G4DormandPrince745( pEquation, nvar );

auto interpDriver= new InterpolationDriver(stepMinimum, dopri5stepper,
                   dopri5stepper->GetNumberOfVariables() );
auto pChordFinder = new G4ChordFinder( interpDriver );
fieldManager->SetChordFinder( pChordFinder );
```

GEANT4 10.4 introduced the capability to use RK methods with the 'First Same as Last' (FSAL) property. Embedded steppers with this property evaluate the field and the derivative in the equation of motion at the endpoint of each step, as an intrinsic part of the method. As a result, after a successful integration step, (one in which the estimated error was acceptable) the derivative at the start of the next step is already available. So one evaluation of the field is saved for every successive integration interval after the first one in each tracking/physical step.

Since GEANT4 10.6 an FSAL capable stepper can be selected for magnetic fields simply by requesting it when constructing a `G4ChordFinder`:

```
G4MagneticField * pMagField;
G4double  stepMinimum = 0.03 * millimeter;
G4int     useFSALstp= 1;

auto pChordFinder= new G4ChordFinder( pMagField, stepMinimum, nullptr, useFSALstp );
fieldManager->SetChordFinder( pChordFinder );
```

### Handling very long steps by switching to helix based stepper

Very long steps of lower energy charged particles can cause excessive simulation time when regular Runge-Kutta methods are used – as these can integrate only a limited angle of a helical track in a single integration step.

This can be a problem for setups in which there is a significant fraction of tracks of low-energy charged particles in a volume with vaccum or a thin gas. In addition integration slowdown or abandoned tracks can occur when muons are tracked in a large air volume with even a fringe magnetic field.

For these setups an alternative type of driver specialised for pure magnetic fields was created. It combines an Interpolation stepper / driver for 'shorter' steps, and a helix-based method for 'long' steps.

It samples the magnetic field at the start of a step, and selects the 'long' step integration method if the helix angle exceed the threshold, currently fixed at 2 `pi`.

This type of driver `G4BFieldIntegrationDriver` was the default driver created by `G4ChordFinder` in Geant4 10.6 for pure magnetic fields.

```
G4MagneticField * pMagField;
G4double  stepMinimum = 0.03 * millimeter;
G4bool    useFSALstp= false;

auto pChordFinder= new G4ChordFinder( pMagField, stepMinimum, nullptr, useFSALstp );
fieldManager->SetChordFinder( pChordFinder );
```

In Geant4 10.7 the default has changed to use an interpolation driver with templated steppers (see next subsection).

As a result, to select it `G4BFieldIntegrationDriver` in Geant4 10.7 a user must use:

```
G4int     driverId = 3;  // B-Field driver = 3
auto pChordFinder= new G4ChordFinder( pMagField, stepMinimum, nullptr, driverId );
fieldManager->SetChordFinder( pChordFinder );
```

It can also be created directly

```
using SmallStepDriver = G4InterpolationDriver<G4DormandPrince745>;
using LargeStepDriver = G4IntegrationDriver<G4HelixHeum>;

auto  regularStepper = new G4DormandPrince745(pEquation);
int   numVar = regularStepper->GetNumberOfVariables();

auto longStepper = std::unique_ptr<G4HelixHeum>(new G4HelixHeum(pEquation));

G4VIntegrationDriver driver =
  new G4BFieldIntegrationDriver(
```

(continues on next page)

(continued from previous page)

```
        std::unique_ptr<SmallStepDriver>(new SmallStepDriver(stepMinimum,
                regularStepper,    numVar ) )
        std::unique_ptr<LargeStepDriver>(new LargeStepDriver(stepMinimum,
                longStepper.get(), numVar ) ) );
```

## Speeding up using steppers templated on equation

To reduce the CPU time of field propagation in Geant4 10.7 an equation class for magnetic fields and some templated stepper classes were created.

The first optimisation is that the templated equation knows the type of the field class:

```
using Equation_t = G4TMagFieldEquation<Field_class_type>;
Equation_t*  equation= new Equation_t(field_object);
```

Given the field class' type, the equation will invoke the field without a virtual call. The relevant include files are:

```
#include "G4TMagFieldEquation.hh"
#include "G4TDormandPrince45.hh"

// For field definition
#include "G4SystemOfUnits.hh"
#include "G4QuadrupoleMagField.hh"
```

To create the templated equation it is simpler to declare each class' type first:

```
using Field_t = G4QuadrupoleMagField;
using Equation_t = G4TMagFieldEquation<Field_t>;

Field_t*     quadMagField = new G4QuadrupoleMagField(1.*tesla/(1.*meter));

Equation_t*  equation=  new Equation_t(quadMagField);
```

but it can also be created directly such as

```
auto  equation= new G4TMagFieldEquation<G4QuadrupoleMagField>(quadMagField);
```

The second optimisation makes the type of the equation known to the templated stepper. This avoid a virtual call from the stepper to the equation.

```
using TemplatedStepper_t = G4TDormandPrince45<Equation_t>;

TemplatedStepper_t* dopri5_stepper=
   new G4TDormandPrince45<Equation_t>( equation );
```

Here `G4TDormandPrince45` is the templated version of the recommended `G4DormandPrince745` stepper that implements the well known 4th/5th order embedded Runge-Kutta method of Dormand and Prince.

Alternative methods which have a templated stepper from Geant4 10.7 include the embedded method of Cash and Karp (`G4TCashKarpRKF45`), and a number of methods which use bisection for error estimatios: the original method of Runge and Kutta (`G4TClassicalRK4`) which used to be the default before Geant4 release 10.4, and two lower order methods, the 2nd order `G4TSimpleRunge` and the third order `G4TSimpleHeum`.

As a further optimisation, it can also be used with a templated driver

```
double stepMin= 0.1 * CLHEP::millimeter;

auto dp45driver =
  new G4IntegrationDriver<TemplatedStepper_t>(stepMin, dopri5_stepper);
```

(continues on next page)

(continued from previous page)

```
auto interpolatingDrv =
  new G4InterpolationDriver<TemplatedStepper_t>(stepMin, dopri5_stepper);
```

which is simplified greatly by using a name for the type of the templated stepper.

Note that the templated stepper class also can be used without using an equation templated on the type of field. This could be preferable in the case of complex field classes in which a large amount of code calculates the value of the field.

A templated stepper also using an alternative type of field. We demonstrate using a full electromagnetic field (which in practice will be a derived class for a user's application) and its equation G4EqMagElectricField:

```
#include "G4EqMagElectricField.hh"

using Equation_t = G4EqMagElectricField;
constexpr nvar= 8; //  Equation integrates over x, p, t

using TemplatedDoPri5_t = G4TDormandPrince45<Equation_t,nvar>;

MyElectroMagneticField *emField= ...; // deriving from

auto emEquation= new G4EqMagElectricField(emField);

TemplatedDoPri5_t* pStepperTDP45 = new TemplatedDoPri5_t( emEquation, nvar );

auto ck45Stepper = new G4TCashKarpRKF45<Equation_t,nvar>( emEquation, nvar );
```

### Using different FSAL steppers – without interpolation

A different method which only has the FSAL property can be used for a magnetic field by adding a flag to the constructor of G4ChordFinder:

```
G4MagneticField * pMagField;
G4double   stepMinimum = 0.03 * millimeter;
G4bool     useFSALstp= true;

auto pChordF= new G4ChordFinder( pMagField, stepMinimum, nullptr, useFSALstp );
```

This uses the constructor:

```
G4ChordFinder::G4ChordFinder( G4MagneticField*        theMagField,
                              G4double                stepMinimum,
                              G4MagIntegratorStepper* pItsStepper,
                              G4bool                  useFSALstepper );
```

Steppers of this type can also be created directly:

Listing 4.14: How to create an 'FSAL'-type stepper

```
#include "G4RK547FEq1.hh"
#include "G4SystemsOfUnits.hh"

using FsalStepperType = G4RK547FEq1;
G4double   stepMinimum= 0.1 * millimeter;   // Minimum size of step (for driver)
int  nvar= 6; // or 8 to include time, Energy

auto fsalStepper=  new FsalStepperType(pEquation, nvar);
```

They must then be coupled to a new type of driver `G4FSALIntegrationDriver` in order to be used in integration. ( This is a derived class of the new base class for drivers `G4VIntegrationDriver`.)

```
auto intgrDriver = new
     G4FSALIntegrationDriver<NewFsalStepperType>(stepMinimum,
                                                 fsalStepper,
                                                 fsalStepper->GetNumberOfVariables() );
```

This also demonstrates one of a new family of (FSAL) steppers `G4RK547FEq1` ( others are `G4RK547FEq2` and `G4RK547FEq3`) which were created to provide an improved equilibrium in integration. When a integration step fails due to an error above threshold, for some setups there can be oscillations in step size that cause multiple bounces between a successful and a failed step. The coefficients of these steppers were optimised to reduce these oscillations, and thus increase the success rate of steps. Initial tests demonstrated a small potential (1.0-2.5%) performance advantage.

### Further reducing the number of field calls to speed-up simulation

An additional method to reduce the number of field evaluations is to avoid recalculating the field value inside a sphere of a given radius `distConst` from the previously evaluated location.

This can be done by utilising the class G4CachedMagneticField class, for the case of pure magnetic fields which do not vary with time.

```
G4MagneticField * pMagField;  // Your field - Defined elsewhere

G4double          distConst = 2.5 * millimeter;
G4MagneticField * pCachedMagField= new G4CachedMagneticField(  pMagField,  distConst);
```

This is not recommended if there are locations in the setup in which large variations occur in the field vector. In those cases it is best to rely on advanced integration method. In particular the Interpolation capabilities introduced in Geant4 10.6 greatly reduced the number of field calls already.

### Choosing different accuracies for the same volume

It is possible to create a `G4FieldManager` which has different properties for particles of different momenta (or depending on other parameters of a track). This is useful, for example, in obtaining high accuracy for 'important' tracks (e.g. muons) and accept less accuracy in tracking other tracks (e.g. electrons). To use this, you must create your own field manager class, derived from `G4FieldManager` which implements the method

```
void ConfigureForTrack( const G4Track * ) override final;
```

and uses it to configure itself using the parameters of the current track.

For example to choose different values for the relative accuracy of integration for particles with energy below or above 2.5 MeV, this could be achieve as follows:

```
class MyFieldManager : G4FieldManager
{
  MyFieldManager() = default;
  ~MyFieldManager() = default;

  void ConfigureForTrack( const G4Track * ) override final;
};

void MyFieldManager::ConfigureForTrack( const G4Track *pTrack)
{
  const G4double lowEepsMin= 1.0e-5, lowEepsMax= 1.0e-4;
  const G4double  hiEepsMin= 2.5e-6,  hiEepsMax= 1.0e-5;

  if( pTrack->GetKineticEnergy() < 2.5 * MeV ) {
     SetMinimumEpsilonStep( lowEepsMin );
     SetMaximumEpsilonStep( lowEepsMax );  // Max relative accuracy
```

(continues on next page)

(continued from previous page)

```
  } else {
     SetMinimumEpsilonStep(  hiEepsMin );
     SetMaximumEpsilonStep(  hiEepsMax );  // Max relative accuracy
  }
}
```

### Parameters that must scale with problem size

The default settings of this module are for problems with the physical size of a typical high energy physics setup, that is, distances smaller than about one kilometer. A few parameters are necessary to carry this information to the magnetic field module, and must typically be rescaled for problems of vastly different sizes in order to get reasonable performance and robustness. Two of these parameters are the maximum acceptable step and the minimum step size.

The **maximum acceptable step** should be set to a distance larger than the biggest reasonable step. If the apparatus in a setup has a diameter of two meters, a likely maximum acceptable steplength would be 10 meters. A particle could then take large spiral steps, but would not attempt to take, for example, a 1000-meter-long step in the case of a very low-density material. Similarly, for problems of a planetary scale, such as the earth with its radius of roughly 6400 km, a maximum acceptable steplength of a few times this value would be reasonable.

An upper limit for the size of a step is a parameter of `G4PropagatorInField`, and can be set by calling its `SetLargestAcceptableStep` method.

The **minimum step size** is used during integration to limit the amount of work in difficult cases. It is possible that strong fields or integration problems can force the integrator to try very small steps; this parameter stops them from becoming unnecessarily small.

Trial steps smaller than this parameter will be treated with less accuracy, and may even be ignored, depending on the situation.

The minimum step size is a parameter of the MagInt_Driver, but can be set in the constructor of G4ChordFinder, as in the source listing above.

### Known Issues

For most integration method it is computationally expensive to change the *miss distance* to very small values, as it causes tracks to be limited to curved sections whose 'sagitta' is smaller than this value. (The sagitta is the distance of the mid-point from the chord between endpoints see e.g. <https://en.wikipedia.org/wiki/Sagitta_%28geometry%29> .) For tracks with small curvature (typically low momentum particles in strong fields) this can cause a large number of steps

- even in areas where there are no volumes to intersect (where the safety could be utilized to partially alleviate this limitation)
- especially in a region near a volume boundary (in which case it is necessary in order to discover whether a track might intersect a volume for only a short distance.)

Requiring such precision for the intersection of all potential volumes is clearly expensive, and in some cases it is not possible to reduce expense greatly.

By contrast, changing the intersection parameter *delta intersection* is less computationally expensive. It causes further calculation for only a fraction of the steps, those that intersect a volume boundary.

### 4.3.3 Spin Tracking

The effects of a particle's motion on the precession of its spin angular momentum in slowly varying external fields are simulated. The relativistic equation of motion for spin is known as the BMT equation. The equation demonstrates a remarkable property; in a purely magnetic field, in vacuum, and neglecting small anomalous magnetic moments, the particle's spin precesses in such a manner that the longitudinal polarization remains a constant, whatever the motion of the particle. But when the particle interacts with electric fields of the medium and multiple scatters, the spin, which is related to the particle's magnetic moment, does not participate, and the need thus arises to propagate it independent of the momentum vector. In the case of a polarized muon beam, for example, it is important to predict the muon's spin direction at decay-time in order to simulate the decay electron (Michel) distribution correctly.

In order to track the spin of a particle in a magnetic field, you need to code the following:

1. in your DetectorConstruction:

   ```
   #include "G4Mag_SpinEqRhs.hh"

   G4Mag_EqRhs* pEquation = new G4Mag_SpinEqRhs(magField);
   G4MagIntegratorStepper* pStepper = new G4ClassicalRK4(pEquation,12);
                                                //    notice the 12
   ```

2. in your PrimaryGenerator:

   ```
   particleGun->SetParticlePolarization(G4ThreeVector p)
   ```

   for example:

   ```
   particleGun->
   SetParticlePolarization(-(particleGun->GetParticleMomentumDirection()));

   // or
   particleGun->
   SetParticlePolarization(particleGun->GetParticleMomentumDirection()
                                         .cross(G4ThreeVector(0.,1.,0.)));
   ```

   where you set the initial spin direction.

While the G4Mag_SpinEqRhs class constructor:

```
G4Mag_SpinEqRhs::G4Mag_SpinEqRhs( G4MagneticField* MagField )
   : G4Mag_EqRhs( MagField )
{
    anomaly = 1.165923e-3;
}
```

sets the muon anomaly by default, the class also comes with the public method:

```
inline void SetAnomaly(G4double a) { anomaly = a; }
```

with which you can set the magnetic anomaly to any value you require.

The code has been rewritten (in Release 9.5) such that field tracking of the spin can now be done for charged and neutral particles with a magnetic moment, for example spin tracking of ultra cold neutrons. This requires the user to set `EnableUseMagneticMoment`, a method of the `G4Transportation` process. The force resulting from the term, MUSDOTNABLAB, is not yet implemented in GEANT4 (for example, simulated trajectory of a neutral hydrogen atom trapped by its magnetic moment in a gradient B-field.)

### 4.3.4 Alternative Integration Methods

There are three alternative integration methods available in Geant4. One is general, can be applied for any equation of motion.

The final, Symplectic Integration, aims to preserve phase space volume and energy exactly – and its implementations achieve this to the order of the method used (currently 2nd order.)

### 4.3.5 Quantum State Simulation

The Quantum State Simulation method is a general integration method which uses polynomial approximation of the solutions of ODEs. In Geant4 its implementation is currently specialised for pure magnetic fields only. It is of potential interest to applications with a large number of volume boundary crossings per track.

Listing 4.15: The simplest way to enable the Quantum State Simulation (QSS) integration method

```
#include "G4ChordFinder.hh"
#include "G4QSSDriverCreator.hh"

{
  G4MagneticField* magField = ...
  G4double  stepMinimum= 0.01 * CLHEP::mm

  auto chordFnd = new G4ChordFinder( magField, stepMinimum, nullptr, kQss2DriverType);
}
..
```

Listing 4.16: How to create a driver for the Quantum State Simulation (QSS) integration method

```
#include "G4QSSDriverCreator.hh"
#include "G4ChordFinder.hh"

{
    auto qssStepper2 = G4QSSDriverCreator::CreateQss2Stepper(pEquation);
    fIntgrDriver = G4QSSDriverCreator::CreateDriver(qssStepper2);

    G4cout << "-- Created QSS driver for B-field integration" << G4endl;

    auto chordFinder= new G4ChordFinder( driver );
}
..
```

### 4.3.6 Bulirsch-Stoer

The Bulirsch-Stoer method is an alternative to Runge-Kutta methods, and uses a midpoint method to obtain an estimate of the trajectory solution. It can be used with any equation of motion. One key use is to cross check results obtained with Runge-Kutta methods.

The current method is fourth order. (tbc)

Listing 4.17: How to create a driver for the Bulirsch Stoer midpoint method

```
#include "G4BulirschStoer.hh"
#include "G4BulirschStoerDriver.hh"

```

(continues on next page)

(continued from previous page)

```
void DetectorConstruction::ConstructSDandField()
{
   G4MagneticField     *magField = new G4UniformMagField( G4ThreeVector( 0, 0, 3.8*tesla );
   G4EquationOfMotion  pEquation= new G4Mag_UsualEqRhs(pMyMagField);

   G4BulirschStoer   * pBSstepper = new G4BulirschStoer( pEquation, nVar, epsilon );
   G4VIntegrationDriver*  driver = new G4IntegrationDriver<G4BulirschStoer>( stepMinimum,␣
↪pBSstepper, nVar );
   G4cout << "Using Bulirsch Stoer method (and driver) - alternative to RK"  << G4endl;

   G4ChordFinder*   chordFinder= new G4ChordFinder( driver );

   // Use this for the global field
   G4FieldMananger* globalFieldMgr= G4TransportationManager::GetTransportationManager()->
↪GetFieldManager();
   globalFieldMgr->SetChordFinder( chordFinder );
}
..
```

### 4.3.7 Symplectic Integration

Some accelerator applications require long-term stability in the integration of energy and/or the preservation of phase-space volume. These are not well served by (medium order) Runge-Kutta method, and even high-order Runge-Kutta methods typically are not able to provide the required accuracy.

To address such applications Geant4 release 11.1 introduces a new integration method, the symplectic 2nd order `Boris` integration method. This method is implemented by the integration driver `G4BorisDriver`.

Listing 4.18: How to create a symplectic 2nd-order Boris integration driver

```
#include "G4BorisScheme.hh"
#include "G4BorisDriver.hh"

void
CreateBorisDriver(G4EquationOfMotion* equation,
                  G4FieldManager* fieldManager,
                  G4double minimumStep
                 )
{
  //   1. Create Scheme and Driver
  auto borisScheme = new G4BorisScheme(equation);
  auto driver = new G4BorisDriver(minimumStep, borisScheme);

  //  2. Create ChordFinder
  auto chordFinder = new G4ChordFinder( driver );

  //  3. Updating Field Manager (with ChordFinder, field)
  fieldManager->SetChordFinder( chordFinder );
}


..
```

Recall that in a pure magnetic field the transportation process, e.g. `G4Transportation`, will conserve energy by ignoring integration errors. The field manager stores a property 'FieldChangesEnergy' which recalls this, and whose default behaviour must be overriden. If investigating energy conservation for a pure magnetic field, you must switch this off by calling the method `SetFieldChangesEnergy(G4bool);` with the argnument `true` for the relevant field manager.

This is demonstrated in the `field01` extended example for the global field manager using:

```
GetGlobalFieldManager()->SetFieldChangesEnergy(true);
```

This enables you to test the level of energy conservation in the integration.

Note in a combined electro-magnetic field this property is automatically `false`, and changes due to integration already affect the particle energy.

## 4.4 Hits

### 4.4.1 Hit

A hit is a snapshot of the physical interaction of a track in the sensitive region of a detector. In it you can store information associated with a `G4Step` object. This information can be

- the position and time of the step,
- the momentum and energy of the track,
- the energy deposition of the step,
- geometrical information,

or any combination of the above.

#### `G4VHit`

`G4VHit` is an abstract base class which represents a hit. You must inherit this base class and derive your own concrete hit class(es). The member data of your concrete hit class can be, and should be, your choice.

As with `G4THitsCollection`, authors of subclasses must declare templated `G4Allocators` for their class. They must also implement *operator new()* and *operator delete()* which use these allocators.

`G4VHit` has two virtual methods, `Draw()` and `Print()`. To draw or print out your concrete hits, these methods should be implemented. How to define the drawing method is described in *Polylines, Markers and Text*.

#### `G4THitsCollection`

`G4VHit` is an abstract class from which you derive your own concrete classes. During the processing of a given event, represented by a `G4Event` object, many objects of the hit class will be produced, collected and associated with the event. Therefore, for each concrete hit class you must also prepare a concrete class derived from `G4VHitsCollection`, an abstract class which represents a vector collection of user defined hits.

`G4THitsCollection` is a template class derived from `G4VHitsCollection`, and the concrete hit collection class of a particular `G4VHit` concrete class can be instantiated from this template class. Each object of a hit collection must have a unique name for each event.

`G4Event` has a `G4HCofThisEvent` class object, that is a container class of collections of hits. Hit collections are stored by their pointers, whose type is that of the base class.

### An example of a concrete hit class

Listing 4.19 shows an example of a concrete hit class.

Listing 4.19: An example of a concrete hit class.

```
//============ header file =====================

#ifndef B2TrackerHit_h
#define B2TrackerHit_h 1

#include "G4VHit.hh"
#include "G4THitsCollection.hh"
#include "G4Allocator.hh"
#include "G4ThreeVector.hh"
#include "tls.hh"

namespace B2
{

/// Tracker hit class
///
/// It defines data members to store the trackID, chamberNb, energy deposit,
/// and position of charged particles in a selected volume:
/// - fTrackID, fChamberNB, fEdep, fPos

class TrackerHit : public G4VHit
{
  public:
    TrackerHit();
    TrackerHit(const TrackerHit&) = default;
    ~TrackerHit() override;

    // operators
    TrackerHit& operator=(const TrackerHit&) = default;
    G4bool operator==(const TrackerHit&) const;

    inline void* operator new(size_t);
    inline void  operator delete(void*);

    // methods from base class
    void Draw() override;
    void Print() override;

    // Set methods
    void SetTrackID  (G4int track)      { fTrackID = track; };
    void SetChamberNb(G4int chamb)      { fChamberNb = chamb; };
    void SetEdep     (G4double de)      { fEdep = de; };
    void SetPos      (G4ThreeVector xyz){ fPos = xyz; };

    // Get methods
    G4int GetTrackID() const     { return fTrackID; };
    G4int GetChamberNb() const   { return fChamberNb; };
    G4double GetEdep() const     { return fEdep; };
    G4ThreeVector GetPos() const { return fPos; };

  private:
    G4int         fTrackID = -1;
    G4int         fChamberNb = -1;
    G4double      fEdep = 0.;
    G4ThreeVector fPos;
};

//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo........oooOO0OOooo......

typedef G4THitsCollection<TrackerHit> TrackerHitsCollection;

extern G4ThreadLocal G4Allocator<TrackerHit>* TrackerHitAllocator;

//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo........oooOO0OOooo......
```

(continues on next page)

(continued from previous page)

```
inline void* TrackerHit::operator new(size_t)
{
  if(!TrackerHitAllocator)
      TrackerHitAllocator = new G4Allocator<TrackerHit>;
  return (void *) TrackerHitAllocator->MallocSingle();
}

//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo........oooOO0OOooo......

inline void TrackerHit::operator delete(void *hit)
{
  TrackerHitAllocator->FreeSingle((TrackerHit*) hit);
}

//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo........oooOO0OOooo......

}

#endif

//============ source file =====================

#include "TrackerHit.hh"

namespace B2
{

G4ThreadLocal G4Allocator<TrackerHit>* TrackerHit::TrackerHitAllocator = 0;

 ... snipped ...
```

G4Allocator is a class for fast allocation of objects to the heap through the paging mechanism. For details of G4Allocator, refer to *General management classes*. Use of G4Allocator is not mandatory, but it is recommended, especially for users who are not familiar with the C++ memory allocation mechanism or alternative tools of memory allocation. On the other hand, note that G4Allocator is to be used **only** for the concrete class that is **not** used as a base class of any other classes. For example, do **not** use the G4Trajectory class as a base class for a customized trajectory class, since G4Trajectory uses G4Allocator.

#### G4THitsMap

G4THitsMap is an alternative to G4THitsCollection. G4THitsMap does not demand G4VHit, but instead any variable which can be mapped with an integer key. Typically the key is a copy number of the volume, and the mapped value could for example be a double, such as the energy deposition in a volume. G4THitsMap is convenient for applications which do not need to output event-by-event data but instead just accumulate them. All the G4VPrimitiveScorer classes discussed in *G4MultiFunctionalDetector and G4VPrimitiveScorer* use G4THitsMap.

G4THitsMap is derived from the G4VHitsCollection abstract base class and all objects of this class are also stored in G4HCofThisEvent at the end of an event. How to access a G4THitsMap object is discussed in the following section (*G4MultiFunctionalDetector and G4VPrimitiveScorer*).

### 4.4.2 Sensitive detector

#### `G4VSensitiveDetector`

`G4VSensitiveDetector` is an abstract base class which represents a detector. The principal mandate of a sensitive detector is the construction of hit objects using information from steps along a particle track. The `ProcessHits()` method of `G4VSensitiveDetector` performs this task using `G4Step` objects as input. The second argument of `ProcessHits()` method, i.e. `G4TouchableHistory`, is obsolete and not used. If user needs to define an artificial second geometry, use *Parallel Geometries*.

`ProcessHits()` method has a return type of `G4bool`. This return value is not used by Geant4 kernel. This return value may be used by the user's use-case where one sensitive detector dispatches `ProcessHits()` to some subsequent (i.e. child) sensitive detectors, and to avoid double-counting, one of these child detector may return true or false.

Your concrete detector class should be instantiated with the unique name of your detector. The name can be associated with one or more global names with "/" as a delimiter for categorizing your detectors. For example

```
myEMcal = new MyEMcal("/myDet/myCal/myEMcal");
```

where `myEMcal` is the name of your detector. The detector must be constructed in `G4VUserDetectorConstruction::ConstructSDandField()` method. It must be assigned to one or more `G4LogicalVolume` objects to set the sensitivity of these volumes. Such assignment must be made in the same `G4VUserDetectorConstruction::ConstructSDandField()` method. The pointer should also be registered to `G4SDManager`, as described in *G4SDManager*.

`G4VSensitiveDetector` has three major virtual methods.

**`ProcessHits()`** This method is invoked by `G4SteppingManager` when a step is composed in the `G4LogicalVolume` which has the pointer to this sensitive detector. The first argument of this method is a `G4Step` object of the current step. The second argument is a `G4TouchableHistory` object for the `Readout geometry` described in the next section. The second argument is `NULL` if `Readout geometry` is not assigned to this sensitive detector. In this method, one or more `G4VHit` objects should be constructed if the current step is meaningful for your detector.

**`Initialize()`** This method is invoked at the beginning of each event. The argument of this method is an object of the `G4HCofThisEvent` class. Hit collections, where hits produced in this particular event are stored, can be associated with the `G4HCofThisEvent` object in this method. The hit collections associated with the `G4HCofThisEvent` object during this method can be used for `during the event processing` digitization.

**`EndOfEvent()`** This method is invoked at the end of each event. The argument of this method is the same object as the previous method. Hit collections occasionally created in your sensitive detector can be associated with the `G4HCofThisEvent` object.

### 4.4.3 G4SDManager

`G4SDManager` is the singleton manager class for sensitive detectors.

#### Activation/inactivation of sensitive detectors

The user interface commands `activate` and `inactivate` are available to control your sensitive detectors. For example:

```
/hits/activate detector_name
/hits/inactivate detector_name
```

where `detector_name` can be the detector name or the category name.

For example, if your EM calorimeter is named

```
/myDet/myCal/myEMcal
/hits/inactivate myCal
```

will inactivate all detectors belonging to the `myCal` category.

#### Access to the hit collections

Hit collections are accessed for various cases.

- Digitization
- Event filtering in `G4VUserStackingAction`
- “End of event” simple analysis
- Drawing / printing hits

The following is an example of how to access the hit collection of a particular concrete type:

```
G4SDManager* fSDM = G4SDManager::GetSDMpointer();
G4RunManager* fRM = G4RunManager::GetRunManager();
G4int collectionID = fSDM->GetCollectionID("collection_name");
const G4Event* currentEvent = fRM->GetCurrentEvent();
G4HCofThisEvent* HCofEvent = currentEvent->GetHCofThisEvent();
MyHitsCollection* myCollection = (MyHitsCollection*)(HC0fEvent->GetHC(collectionID));
```

### 4.4.4 `G4MultiFunctionalDetector` and `G4VPrimitiveScorer`

`G4MultiFunctionalDetector` is a concrete class derived from `G4VSensitiveDetector`. Instead of implementing a user-specific detector class, `G4MultiFunctionalDetector` allows the user to register `G4VPrimitiveScorer` classes to build up the sensitivity. `G4MultiFunctionalDetector` should be instantiated in the users detector construction with its unique name and should be assigned to one or more `G4LogicalVolumes`.

`G4VPrimitiveScorer` is an abstract base class representing a class to be registered to `G4MultiFunctionalDetector` that creates a `G4THitsMap` object of one physics quantity for an event. GEANT4 provides many concrete primitive scorer classes listed in *Concrete classes of G4VPrimitiveScorer*, and the user can also implement his/her own primitive scorers. Each primitive scorer object must be instantiated with a name that must be unique among primitive scorers registered in a `G4MultiFunctionalDetector`. Please note that a primitive scorer object must **not** be shared by more than one `G4MultiFunctionalDetector` object.

As mentioned in *Hit*, each `G4VPrimitiveScorer` generates one `G4THitsMap` object per event. The name of the map object is the same as the name of the primitive scorer. Each of the concrete primitive scorers listed in

*Concrete classes of G4VPrimitiveScorer* generates a `G4THitsMap<G4double>` that maps a `G4double` value to its key integer number. By default, the key is taken as the copy number of the `G4LogicalVolume` to which `G4MultiFunctionalDetector` is assigned. In case the logical volume is uniquely placed in its mother volume and the mother is replicated, the copy number of its mother volume can be taken by setting the second argument of the `G4VPrimitiveScorer` constructor, "*depth*" to 1, i.e. one level up. Furthermore, in case the key must consider more than one copy number of a different geometry hierarchy, the user can derive his/her own primitive scorer from the provided concrete class and implement the `GetIndex(G4Step*)` virtual method to return the unique key.

Listing 4.20 shows an example of primitive sensitivity class definitions.

Listing 4.20: An example of defining primitive sensitivity classes taken from `RE06DetectorConstruction`.

```
void RE06DetectorConstruction::SetupDetectors()
{
  G4String filterName, particleName;

  G4SDParticleFilter* gammaFilter =
    new G4SDParticleFilter(filterName="gammaFilter",particleName="gamma");
  G4SDParticleFilter* electronFilter =
    new G4SDParticleFilter(filterName="electronFilter",particleName="e-");
  G4SDParticleFilter* positronFilter =
    new G4SDParticleFilter(filterName="positronFilter",particleName="e+");
  G4SDParticleFilter* epFilter = new G4SDParticleFilter(filterName="epFilter");
  epFilter->add(particleName="e-");
  epFilter->add(particleName="e+");


  for(G4int i=0;i<3;i++)
  {
   for(G4int j=0;j<2;j++)
   {
    // Loop counter j = 0 : absorber
    //                = 1 : gap
    G4String detName = fCalName[i];
    if(j==0)
    { detName += "_abs"; }
    else
    { detName += "_gap"; }
    G4MultiFunctionalDetector* det = new G4MultiFunctionalDetector(detName);

    //  The second argument in each primitive means the "level" of geometrical hierarchy,
    // the copy number of that level is used as the key of the G4THitsMap.
    //  For absorber (j = 0), the copy number of its own physical volume is used.
    //  For gap (j = 1), the copy number of its mother physical volume is used, since there
    // is only one physical volume of gap is placed with respect to its mother.
    G4VPrimitiveScorer* primitive;
    primitive = new G4PSEnergyDeposit("eDep",j);
    det->RegisterPrimitive(primitive);
    primitive = new G4PSNofSecondary("nGamma",j);
    primitive->SetFilter(gammaFilter);
    det->RegisterPrimitive(primitive);
    primitive = new G4PSNofSecondary("nElectron",j);
    primitive->SetFilter(electronFilter);
    det->RegisterPrimitive(primitive);
    primitive = new G4PSNofSecondary("nPositron",j);
    primitive->SetFilter(positronFilter);
    det->RegisterPrimitive(primitive);
    primitive = new G4PSMinKinEAtGeneration("minEkinGamma",j);
    primitive->SetFilter(gammaFilter);
    det->RegisterPrimitive(primitive);
    primitive = new G4PSMinKinEAtGeneration("minEkinElectron",j);
    primitive->SetFilter(electronFilter);
    det->RegisterPrimitive(primitive);
    primitive = new G4PSMinKinEAtGeneration("minEkinPositron",j);
```

(continues on next page)

(continued from previous page)

```
    primitive->SetFilter(positronFilter);
    det->RegisterPrimitive(primitive);
    primitive = new G4PSTrackLength("trackLength",j);
    primitive->SetFilter(epFilter);
    det->RegisterPrimitive(primitive);
    primitive = new G4PSNofStep("nStep",j);
    primitive->SetFilter(epFilter);
    det->RegisterPrimitive(primitive);

    G4SDManager::GetSDMpointer()->AddNewDetector(det);
    if(j==0)
    { layerLogical[i]->SetSensitiveDetector(det); }
    else
    { gapLogical[i]->SetSensitiveDetector(det); }
   }
  }
}
```

Each `G4THitsMap` object can be accessed from `G4HCofThisEvent` with a unique collection ID number. This ID number can be obtained from `G4SDManager::GetCollectionID()` with a name of `G4MultiFunctionalDetector` and `G4VPrimitiveScorer` connected with a slush ("/"). `G4THitsMap` has a [] operator taking the key value as an argument and returning **the pointer** of the value. Please note that the [] operator returns **the pointer** of the value. If you get zero from the [] operator, it does **not** mean the value is zero, but that the provided key does not exist. The value itself is accessible with an asterisk ("*"). It is advised to check the validity of the returned pointer before accessing the value. `G4THitsMap` also has a += operator in order to accumulate event data into run data. Listing 4.21 shows the use of `G4THitsMap`.

Listing 4.21: An example of accessing to `G4THitsMap` objects.

```
#include "Run.hh"

#include "G4RunManager.hh"
#include "G4Event.hh"

#include "G4SDManager.hh"
#include "G4HCofThisEvent.hh"
#include "G4THitsMap.hh"
#include "G4SystemOfUnits.hh"

namespace B3b
{

//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo........oooOO0OOooo......

Run::Run()
{ }

//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo........oooOO0OOooo......

Run::~Run()
{ }

//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo........oooOO0OOooo......

void Run::RecordEvent(const G4Event* event)
{
  if ( fCollID_cryst < 0 ) {
   fCollID_cryst
     = G4SDManager::GetSDMpointer()->GetCollectionID("crystal/edep");
   //G4cout << " fCollID_cryst: " << fCollID_cryst << G4endl;
  }

  if ( fCollID_patient < 0 ) {
```

(continues on next page)

(continued from previous page)

```
    fCollID_patient
      = G4SDManager::GetSDMpointer()->GetCollectionID("patient/dose");
    //G4cout << " fCollID_patient: " << fCollID_patient << G4endl;
   }

  G4int evtNb = event->GetEventID();

  if (evtNb%fPrintModulo == 0) {
    G4cout << G4endl << "---> end of event: " << evtNb << G4endl;
  }

  //Hits collections
  //
  G4HCofThisEvent* HCE = event->GetHCofThisEvent();
  if(!HCE) return;

  //Energy in crystals : identify 'good events'
  //
  const G4double eThreshold = 500*keV;
  G4int nbOfFired = 0;

  G4THitsMap<G4double>* evtMap =
    static_cast<G4THitsMap<G4double>*>(HCE->GetHC(fCollID_cryst));

  std::map<G4int,G4double*>::iterator itr;
  for (itr = evtMap->GetMap()->begin(); itr != evtMap->GetMap()->end(); itr++) {
    G4double edep = *(itr->second);
    if (edep > eThreshold) nbOfFired++;
    ///G4int copyNb  = (itr->first);
    ///G4cout << G4endl << "  cryst" << copyNb << ": " << edep/keV << " keV ";
  }
  if (nbOfFired == 2) fGoodEvents++;

  //Dose deposit in patient
  //
  G4double dose = 0.;

  evtMap = static_cast<G4THitsMap<G4double>*>(HCE->GetHC(fCollID_patient));

  for (itr = evtMap->GetMap()->begin(); itr != evtMap->GetMap()->end(); itr++) {
    ///G4int copyNb  = (itr->first);
    dose = *(itr->second);
  }
  fSumDose += dose;
  fStatDose += dose;

  G4Run::RecordEvent(event);
}

//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo........oooOO0OOooo......

void Run::Merge(const G4Run* aRun)
{
  const Run* localRun = static_cast<const Run*>(aRun);
  fGoodEvents += localRun->fGoodEvents;
  fSumDose    += localRun->fSumDose;
  fStatDose   += localRun->fStatDose;
  G4Run::Merge(aRun);
}

//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo........oooOO0OOooo......

}
```

### 4.4.5 Concrete classes of `G4VPrimitiveScorer`

With GEANT4 version 8.0, several concrete primitive scorer classes are provided, all of which are derived from the `G4VPrimitiveScorer` abstract base class and which are to be registered to `G4MultiFunctionalDetector`. Each of them contains one `G4THitsMap` object and scores a simple double value for each key.

#### Track length scorers

**`G4PSTrackLength`** The track length is defined as the sum of step lengths of the particles inside the cell. By default, the track weight is not taken into account, but could be used as a multiplier of each step length if the `Weighted()` method of this class object is invoked.

**`G4PSPassageTrackLength`** The passage track length is the same as the track length in `G4PSTrackLength`, except that only tracks which pass through the volume are taken into account. It means newly-generated or stopped tracks inside the cell are excluded from the calculation. By default, the track weight is not taken into account, but could be used as a multiplier of each step length if the `Weighted()` method of this class object is invoked.

#### Deposited energy scorers

**`G4PSEnergyDeposit`** This scorer stores a sum of particles' energy deposits at each step in the cell. The particle weight is multiplied at each step.

**`G4PSDoseDeposit`** In some cases, dose is a more convenient way to evaluate the effect of energy deposit in a cell than simple deposited energy. The dose deposit is defined by the sum of energy deposits at each step in a cell divided by the mass of the cell. The mass is calculated from the density and volume of the cell taken from the methods of `G4VSolid` and `G4LogicalVolume`. The particle weight is multiplied at each step.

**`G4PSDoseDeposit3D`** In the case of replica or nested geometries it is necessary to determine voxel numbers from within the replica hierarchy. For example if the *z-axis* is parameterised and *y* is replica of *x* then the voxel number needs to be calculated accordingly:

```
G4PSDoseDeposit3D(("DoseDeposit", fNoVoxelsZ, fNoVoxelsY, fNoVoxelsX, 0, 2, 1);
```

The last three arguments are optional, however required to determine the depth according to each axis (or replica direction). The class creates instances of G4PSDoseDeposit according to:

```
i * fNj * fNk + j * fNk + k;
```

where *i*, *j* and *k* correspond to the 3 arguments for number of voxels and replica depth in the declaration.

#### Current and flux scorers

There are two different definitions of a particle's flow for a given geometry. One is a current and the other is a flux. In our scorers, the current is simply defined as the number of particles (with the particle's weight) at a certain surface or volume, while the flux takes the particle's injection angle to the geometry into account. The current and flux are usually defined at a surface, but volume current and volume flux are also provided.

**`G4PSFlatSurfaceCurrent`** Flat surface current is a surface based scorer. The present implementation is limited to scoring only at the -Z surface of a `G4Box` solid. The quantity is defined by the number of tracks that reach the surface. The user must choose a direction of the particle to be scored. The choices are fCurrent_In, fCurrent_Out, or fCurrent_InOut, one of which must be entered as the second argument of the constructor. Here, fCurrent_In scores incoming particles to the cell, while fCurrent_Out scores only outgoing particles from the cell. fCurrent_InOut scores both directions. The current is multiplied by particle weight and is normalized for a unit area.

**G4PSSphereSurfaceCurrent** Sphere surface current is a surface based scorer, and similar to the G4PSFlatSurfaceCurrent. The only difference is that the surface is defined at the **inner surface** of a G4Sphere solid.

**G4PSPassageCurrent** Passage current is a volume-based scorer. The current is defined by the number of tracks that pass through the volume. A particle weight is applied at the exit point. A passage current is defined for a volume.

**G4PSFlatSurfaceFlux** Flat surface flux is a surface based flux scorer. The surface flux is defined by the number of tracks that reach the surface. The expression of surface flux is given by the sum of W/cos(t)/A, where W, t and A represent particle weight, injection angle of particle with respect to the surface normal, and area of the surface. The user must enter one of the particle directions, fFlux_In, fFlux_Out, or fFlux_InOut in the constructor. Here, fFlux_In scores incoming particles to the cell, while fFlux_Out scores outgoing particles from the cell. fFlux_InOut scores both directions.

**G4PSCellFlux** Cell flux is a volume based flux scorer. The cell flux is defined by a track length (L) of the particle inside a volume divided by the volume (V) of this cell. The track length is calculated by a sum of the step lengths in the cell. The expression for cell flux is given by the sum of (W*L)/V, where W is a particle weight, and is multiplied by the track length at each step.

**G4PSPassageCellFlux** Passage cell flux is a volume based scorer similar to `G4PSCellFlux`. The only difference is that tracks which pass through a cell are taken into account. It means generated or stopped tracks inside the volume are excluded from the calculation.

### Other scorers

**G4PSMinKinEAtGeneration** This scorer records the minimum kinetic energy of secondary particles at their production point in the volume in an event. This primitive scorer does not integrate the quantity, but records the minimum quantity.

**G4PSNofSecondary** This class scores the number of secondary particles generated in the volume. The weight of the secondary track is taken into account.

**G4PSNofStep** This class scores the number of steps in the cell. A particle weight is not applied.

**G4PSCellCharge** This class scored the total charge of particles which has stopped in the volume.

## 4.4.6 G4VSDFilter and its derived classes

`G4VSDFilter` is an abstract class that represents a track filter to be associated with `G4VSensitiveDetector` or `G4VPrimitiveScorer`. It defines a virtual method

```
G4bool Accept(const G4Step*)
```

that should return *true* if this particular step should be scored by the `G4VSensitiveDetector` or `G4VPrimitiveScorer`.

While the user can implement his/her own filter class, GEANT4 version 8.0 provides the following concrete filter classes:

**G4SDChargedFilter** All charged particles are accepted.

**G4SDNeutralFilter** All neutral particles are accepted.

**G4SDParticleFilter** Particle species which are registered to this filter object by `Add("particle_name")` are accepted. More than one species can be registered.

**G4SDKineticEnergyFilter** A track with kinetic energy greater than or equal to EKmin and smaller than EKmin is accepted. EKmin and EKmax should be defined as arguments of the constructor. The default values of EKmin and EKmax are zero and DBL_MAX.

**G4SDParticleWithEnergyFilter** Combination of `G4SDParticleFilter` and `G4SDParticleWithEnergyFilter`.

The use of the `G4SDParticleFilter` class is demonstrated in Listing 4.20, where filters which accept gamma, electron, positron and electron/positron are defined.

### 4.4.7 Multiple sensitive detectors associated to a single logical-volume

From GEANT4 Version 10.3 it is possible to attach multiple sensitive detectors to a single geometrical element. This is achieved via the use of a special proxy class, to which multiple child sensitive detectors are attached: `G4MultiSensitiveDetector`. The kernel still sees a single sensitive detector associated to any given logical-volume, but the proxy will dispatch the calls from kernel to all the attached child sensitive detectors.

When using the `G4VUserDetectorConstruction::SetSensitiveDetector(...)` utility method the handling of multiple sensitive detectors is done automatically. Multiple calls to the method passing the same logical volume will trigger the creation and setup of an instance of `G4MultiSensitiveDetector`.

For more complex use cases it may be necessary to manually instantiate and setup an instance of `G4MultiSensitiveDetector`. For this advanced use case you can refer to the implementation of the `G4VUserDetectorConstruction::SetSensitiveDetector(G4LogicalVolume* logVol, G4VSensitiveDetector* aSD)` utility method.

Listing 4.22: An example of the use of `G4MultiSensitiveDetector`.

```
void MyDetectorConstruction::ConstructSDandField()
{
  auto sdman = G4SDManager::GetSDMpointer();
  //...
  auto mySD = new mySD1("/SD1");
  sdman->AddNewDetector(mySD);//Note we explicitly add the SD to the manager
  SetSensitiveDetector("LogVolName",mySD);
  auto mySD2 = new MySD2("/SD2");
  sdman->AddNewDetector(mySD2);
  //This will trigger automatic creation and setup of proxy
  SetSensitiveDetector("LogVolName",mySD2);
  //...
}
```

### 4.4.8 Utilities

#### UI-command base scoring

Command-based scoring functionality offers the user a possibility to define a scoring-mesh and various scorers for commonly-used physics quantities such as dose, flux, etc. via UI commands.

Due to small performance overhead, it does not come by default. To use this functionality, `G4ScoringManager` has to be activated after the instantiation of G4RunManager in the `main()` function, see Listing 4.23. This will create the UI commands in /score directory.

Listing 4.23: Activation of UI-command base scoring in `main()`

```
#include "G4ScoringManager.hh"
int main()
{
 // ...
 G4RunManager* runManager = new G4RunManager;
 G4ScoringManager::GetScoringManager();
 // ...
}
```

An example of a macro creating a scoring mesh of box type with two scorers and a filter is given below:

```
# Define scoring mesh
/score/create/boxMesh boxMesh_1
/score/mesh/boxSize 100. 100. 100. cm
/score/mesh/nBin 30 30 30

# Define scoring quantity
/score/quantity/energyDeposit boxMash keV

# Define a filter
/score/filter/charged

# Close mesh
/score/close
```

Detailed usage of command-based scoring is given in the section *Command-based scoring*.

`G4ScoringManager` provides also a default score writer which dumps every entry of one quantity of a mesh for all quantities of the mesh one by one in CSV format. To alternate the file format the user can implement his/her own score writer deriving from `G4VUserScoreWriter` base class and set it to `G4ScoringManager`. To demonstrate this, `RE03UserScoreWriter` is included in the extended RE03 example in the runAndEvent category.

### Score Ntuple Writer

It is also possible to save the scorers hits using Geant4 analysis tools. This functionality is assured by the `G4VScoreNtupleWriter` interface (since 10.5) and the `G4TScoreNtupleWriter` and `G4TScoreNtupleWriterMessenger` classses (since 10.6).

This feature is demonstrated in the basic examples B3 and B4d. The example of activating the score ntuple writer is given below:

```
#include "G4AnalysisManager.hh"
#include "G4TScoreNtupleWriter.hh"

  // Activate score ntuple writer
  G4TScoreNtupleWriter<G4AnalysisManager> scoreNtupleWriter;
  scoreNtupleWriter.SetVerboseLevel(1);
```

The Geant4 UI commands defined in `G4TScoreNtupleWriterMessenger` can be used to choose the output file name and the level of verbosity:

```
/score/ntuple/writerFileName name
/score/ntuple/writerVerbose 1
```

## 4.5 Digitization

### 4.5.1 Digi

A hit is created by a sensitive detector when a step goes through it. Thus, the sensitive detector is associated to the corresponding `G4LogicalVolume` object(s). On the other hand, a digit is created using information of hits and/or other digits by a digitizer module. The digitizer module is not associated with any volume, and you have to implicitly invoke the `Digitize()` method of your concrete `G4VDigitizerModule` class.

Typical usages of digitizer module include:

- simulate ADC and/or TDC
- simulate readout scheme

- generate raw data
- simulate trigger logics
- simulate pile up

#### G4VDigi

G4VDigi is an abstract base class which represents a digit. You have to inherit this base class and derive your own concrete digit class(es). The member data of your concrete digit class should be defined by yourself. G4VDigi has two virtual methods, Draw() and Print().

As with G4VHit, authors of subclasses must declare templated G4Allocators for their digit class. They must also implement *operator new()* and *operator delete()* which use these allocators.

#### G4TDigiCollection

G4TDigiCollection is a template class for digits collections, which is derived from the abstract base class G4VDigiCollection. G4Event has a G4DCofThisEvent object, which is a container class of collections of digits. The usages of G4VDigi and G4TDigiCollection are almost the same as G4VHit and G4THitsCollection, respectively, explained in the previous section.

As with G4THitsCollection, authors of subclasses must declare templated G4Allocators for their collection class. They must also implement *operator new()* and *operator delete()* which use these allocators.

### 4.5.2 Digitizer module

#### G4VDigitizerModule

G4VDigitizerModule is an abstract base class which represents a digitizer module. It has a pure virtual method, Digitize(). A concrete digitizer module must have an implementation of this virtual method. The GEANT4 kernel classes do not have a "built-in" invocation to the Digitize() method. You have to implement your code to invoke this method of your digitizer module.

In the Digitize() method, you construct your G4VDigi concrete class objects and store them to your G4TDigiCollection concrete class object(s). Your collection(s) should be associated with the G4DCofThisEvent object.

#### G4DigiManager

G4DigiManager is the singleton manager class of the digitizer modules. All of your concrete digitizer modules should be registered to G4DigiManager with their unique names.

```
G4DigiManager * fDM = G4DigiManager::GetDMpointer();
MyDigitizer * myDM = new MyDigitizer( "/myDet/myCal/myEMdigiMod" );
fDM->AddNewModule(myDM);
```

Your concrete digitizer module can be accessed from your code using the unique module name.

```
G4DigiManager * fDM = G4DigiManager::GetDMpointer();
MyDigitizer * myDM = fDM->FindDigitizerModule( "/myDet/myCal/myEMdigiMod" );
myDM->Digitize();
```

Also, G4DigiManager has a Digitize() method which takes the unique module name.

```
G4DigiManager * fDM = G4DigiManager::GetDMpointer();
MyDigitizer * myDM = fDM->Digitize( "/myDet/myCal/myEMdigiMod" );
```

### How to get hitsCollection and/or digiCollection

`G4DigiManager` has the following methods to access to the hits or digi collections of the currently processing event or of previous events.

First, you have to get the collection ID number of the hits or digits collection.

```
G4DigiManager * fDM = G4DigiManager::GetDMpointer();
G4int myHitsCollID = fDM->GetHitsCollectionID( "hits_collection_name" );
G4int myDigiCollID = fDM->GetDigiCollectionID( "digi_collection_name" );
```

Then, you can get the pointer to your concrete `G4THitsCollection` object or `G4TDigiCollection` object of the currently processing event.

```
MyHitsCollection * HC = fDM->GetHitsCollection( myHitsCollID );
MyDigiCollection * DC = fDM->GetDigiCollection( myDigiCollID );
```

In case you want to access to the hits or digits collection of previous events, add the second argument.

```
MyHitsCollection * HC = fDM->GetHitsCollection( myHitsCollID, n );
MyDigiCollection * DC = fDM->GetDigiCollection( myDigiCollID, n );
```

where, `n` indicates the hits or digits collection of the `n`th previous event.

## 4.6 Birks Quenching

The current versions of hadronic string models (FTF & QGS) produce hadronic showers with (a few percent) higher energy response than the stable released (as used in LHC productions) version of these models. Test-beam and collider data seem to indicate lower energy response in hadronic showers than currently provided by Geant4 simulations. This is the main reason why the development versions of the string models were not released in two previous public versions of Geant4 (10.3 and 10.4), in spite of providing an overall better description of thin-target data. We think that the main reason why the simulation overshot the data regarding the energy response of hadronic showers is in fact due to an incorrect treatment of the quenching of the signal - the conversion from the energy deposited by ionizing particles in a sensitive detector to the observed electronic (readout) signal is not linear, with proportionally less signal for higher densities of deposited energy, for both scintillation light and ionization electron-hole/ion pairs. This quenching effect is traditionally described by the simple, phenomenological "law" suggested many years ago by Birks. Its main parameter is fitted from experimental data under the assumption that the observed energy is related to the incident particle energy loss. This does not consider delta-ray production which will result in lower energy deposit (density). As a result of this approximation, the density of deposited energy is overestimated, which implies that the Birks coefficient, as fitted from the experimental data, gets underestimated. Using this Birks coefficient in simulations where delta rays are emitted (and considered discretely), as in practice for all simulations of high-energy experiments, results in underestimating the quenching effect, and therefore predicting larger signals than in reality. The correct Birks coefficient to be used in a simulation depends on the production threshold which is chosen in the simulation, with lower thresholds producing a larger delta-ray component and therefore reducing the density of the energy deposition along the ionizing track, and hence requiring an even higher Birks coefficient. We suggest the following pragmatic approach to incorporating Birks quenching: The calibration of a calorimeter - i.e. the conversion from the electronic signal produced by a shower and the energy of the primary particle that initiates the shower - is typically done for test beam data with an electron of a given energy, e.g. 20 GeV. We suggest to add an extra step to this calibration, in which the Birks coefficient used in the simulation is tuned to reproduce the ratio of the energy response of a hadron (typically a charged pion or a proton) and the energy response of an electron of the same energy (this ratio is indicated as "h/e"). It is natural to consider the same beam energy used for the calibration, e.g. 20 GeV, but in principle it could be a different one; note also that the tuning of the Birks coefficient is independent from the calibration constant, given that the latter cancels out from the ratio h/e. Of course, with the tuning of the Birks coefficient as suggested above we compensate also for some of the intrinsic inaccuracies in the modelling of hadronic interactions; however, this effect is valid uniquely at the energy where the tuning is done (e.g. 20 GeV), and limited only to the energy response. For other observables (energy resolution, longitudinal and lateral shower shapes), and for all other energies, this procedure has a minimal impact, i.e. should not reduce the prediction-power of the simulation.

## 4.7 Object Persistency

### 4.7.1 Persistency in GEANT4

Object persistency is provided by GEANT4 as an optional functionality.

When a usual (transient) object is created in C++, the object is placed onto the application heap and it ceases to exist when the application terminates. Persistent objects, on the other hand, live beyond the termination of the application process and may then be accessed by other processes (in some cases, by processes on other machines).

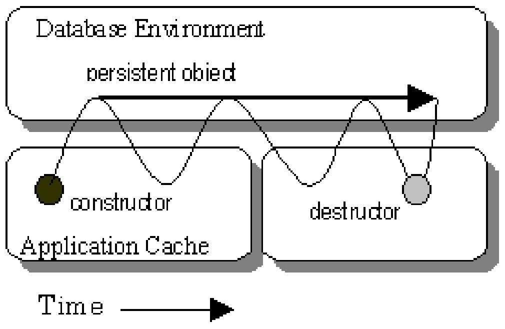


Fig. 4.8: Persistent object.

C++ does not have, as an intrinsic part of the language, the ability to store and retrieve persistent objects. GEANT4 provides an abstract framework for persistency of hits, digits and events.

Two examples demonstrating an implementation of object persistency using one of the tools accessible through the available interface, is provided in `examples/extended/persistency`.

### 4.7.2 Using Root-I/O for persistency of GEANT4 objects

Object persistency of GEANT4 objects is also possible by using the Root-I/O features through Root (since release `v6.04/08`).

The basic steps that one needs to do in order to use Root-I/O for arbitrary C++ classes is:

1. Generate the dictionary for the given classes from Root (this usually is done by adding the appropriate command to the makefile)
2. Add initialization of Root-I/O and loading of the generated dictionary for the given classes in the appropriate part of the code
3. Whenever the objects to be persistified are available, call the `WriteObject` method of `TFile` with the pointer to the appropriate object as argument (usually it is some sort of container, like `std::vector` containing the collection of objects to be persistified)

The two examples (`P01` and `P02`) provided in `examples/extended/persistency` demonstrate how to perform object persistency using the Root-I/O mechanism for storing hits and geometry description.

## 4.8 Parallel Geometries

### 4.8.1 A parallel world

Occasionally, it is not straightforward to define geometries for sensitive detectors, importance geometries or envelopes for shower parameterization to be coherently assigned to volumes in the tracking (mass) geometry. The parallel navigation functionality introduced since release 8.2 of GEANT4, allows the user to define more than one world simultaneously. The `G4CoupledTransportation` process will see all worlds simultaneously; steps will be limited by every boundaries of the mass and parallel geometries. `G4Transportation` is automatically replaced `G4CoupledTransportation`.

In a parallel world, the user can define volumes in arbitrary manner with sensitivity, regions, shower parameterization setups, and/or importance weight for biasing. Volumes in different worlds may overlap.

Any kind of `G4VSensitiveDetector` object can be defined in volumes in a parallel world, exactly at the same manner for the mass geometry. `G4Step` object given as an argument of `ProcessHit()` method contains geometrical information of the associated world.

Here are restrictions to be considered for the parallel geometry:

- Production thresholds and EM field are used only from the mass geometry. Even if such *physical* quantities are defined in a parallel world, they do not affect to the simulation.
- Although all worlds will be comprehensively taken care by the `G4CoupledTransportation` process for the navigation, each parallel world must have its own unique object of `G4ParallelWorldProcess` process (for instance created with `G4ParallelWorldPhysics` constructor registered to a modular physics list).
- Volumes in a parallel world may have materials. Such materials overwrite the materials defined in the mass geometry if the `"layered mass geometry"` switch of the `G4ParallelWorldProcess` constructor is set.

### 4.8.2 Defining a parallel world

A parallel world should be defined in the `Construct()` virtual method of the user's class derived from the abstract base class `G4VUserParallelWorld`. If needed, sensitive detectors must be defined in the `ConstructSD()` method of the same derived class. Please note that EM field cannot be defined in a parallel world.

Listing 4.24: An example header file of a concrete user parallel world class.

```
#ifndef MyParallelWorld_h
#define MyParallelWorld_h 1

#include "globals.hh"
#include "G4VUserParallelWorld.hh"

class MyParallelWorld : public G4VUserParallelWorld
{
  public:
    MyParallelWorld(G4String worldName);
    virtual ~MyParallelWorld();

  public:
    virtual void Construct();
    virtual void ConstructSD();
```

(continues on next page)

(continued from previous page)

```
};

#endif
```

A parallel world must have its unique name, which should be set to the `G4VUserParallelWorld` base class as an argument of the base class constructor.

The world physical volume of the parallel world is provided by the `G4RunManager` as a clone of the mass geometry. In the `Construct()` virtual method of the user's class, the pointer to this cloned world physical volume is available through the `GetWorld()` method defined in the base class. The user should fill the volumes in the parallel world by using this provided world volume. For a logical volume in a parallel world, the material pointer can be `nullptr`. Even if specified a valid material pointer, unless `"layered mass geometry"` switch of the `G4ParallelWorldProcess` constructor is set, it will not be taken into account by any physics process.

Listing 4.25: An example source code of a concrete user parallel world class.

```
#include "MyParallelWorld.hh"
#include "G4LogicalVolume.hh"
#include "G4VPhysicalVolume.hh"
#include "G4Box.hh"
#include "G4PVPlacement.hh"

MyParallelWorld::MyParallelWorld(G4String worldName)
:G4VUserParallelWorld(worldName)
{;}

MyParallelWorld::~MyParallelWorld()
{;}

void MyParallelWorld::Construct()
{
  G4VPhysicalVolume* ghostWorld = GetWorld();
  G4LogicalVolume* worldLogical = ghostWorld->GetLogicalVolume();

  // place volumes in the parallel world here. For example ...
  //
  G4Box * ghostSolid = new G4Box("GhostdBox", 60.*cm, 60.*cm, 60.*cm);
  G4LogicalVolume * ghostLogical
        = new G4LogicalVolume(ghostSolid, 0, "GhostLogical", 0, 0, 0);
  new G4PVPlacement(0, G4ThreeVector(), ghostLogical,
                    "GhostPhysical", worldLogical, 0, 0);
}
```

In case the user needs to define more than one parallel worlds, each of them must be implemented through its dedicated class. Each parallel world should be registered to the mass geometry class using the method `RegisterParallelWorld()` available through the class `G4VUserDetectorConstruction`. The registration must be done before the mass world is registered to the `G4RunManager`. Each parallel world should also have its own `G4ParallelWorldPhysics` constructor registered to the physics list using the method `RegisterPhysics()` available through the class `G4VModularPhysicsList`.

Listing 4.26: Typical implementation in the `main()` to define a parallel world.

```
// RunManager construction
//
G4RunManager* runManager = new G4RunManager;

// mass world
//
MyDetectorConstruction* massWorld = new MyDetectorConstruction;
```

(continues on next page)

(continued from previous page)

```
// parallel world
//
G4String paraWorldName = "ParallelWorld";
massWorld->RegisterParallelWorld(new MyParallelWorld(paraWorldName));

// set mass world to run manager
//
runManager->SetUserInitialization(massWorld);

// physics list
//
G4VModularPhysicsList* physicsList = new FTFP_BERT;
physicsList->RegisterPhysics(new G4ParallelWorldPhysics(paraWorldName));
runManager->SetUserInitialization(physicsList);
```

## 4.8.3 Layered mass geometry

If `"layered mass geometry"` switch of the `G4ParallelWorldProcess` constructor is set, that parallel world is conceptually layered on top of the mass geometry. If more than one parallel worlds are defined, later-defined world comes on top of others. A track will see the material of the top layer, if it is `nullptr`, then one layer beneath. Thus, user has to make sure volumes in a parallel world should have `nullptr` as their materials except for volumes he/she really wants to overwrite.

Listing 4.27: Typical implementation in the `main()` to define a layered mass geometry.

```
// RunManager construction
//
G4RunManager* runManager = new G4RunManager;

// mass world
//
MyDetectorConstruction* massWorld = new MyDetectorConstruction;

// parallel world
//
G4String paraWorldName = "ParallelWorld";
massWorld->RegisterParallelWorld(new MyParallelWorld(paraWorldName));

// set mass world to run manager
//
runManager->SetUserInitialization(massWorld);

// physics list
//
G4VModularPhysicsList* physicsList = new FTFP_BERT;
physicsList->RegisterPhysics(new G4ParallelWorldPhysics(paraWorldName,true));
runManager->SetUserInitialization(physicsList);
```

For an information to advanced users, instead of using `G4ParallelWorldPhysics` physics constructor, once can define `G4ParallelWorldProcess` in his/her physics list and assign it only to some selected kind of particle types. In this case, this parallel world will be seen only by these kinds of particles.

## 4.9 Command-based scoring

### 4.9.1 Introduction

Command-based scoring in GEANT4 defines G4MultiFunctionalDetector to a volume that is either defined in the tracking volume or created in a dedicated parallel world utilizing parallel navigation as described in the previous sections. The parallel world volume can be a scoring mesh or a scoring probe.

Once a scoring volume is defined, through interactive commands, the user can define arbitrary number of primitive scorers to score physics quantities and filters to be associated to each primitive scorer.

After scoring (i.e. a run), the user can dump scores into a file. Scores are automatically merged over worker threads. Also, for scoring mesh, scores can be visualized as well. All available UI commands are listed in List of built-in commands.

Command-based scoring is an optional functionality and the user has to explicitly define its use in the main(). To do this, the method G4ScoringManager::GetScoringManager() must be invoked right after the instantiation of G4RunManager. The scoring manager is a singleton object, and the pointer accessed above should not be deleted by the user.

Listing 4.28: A user main() to use the command-based scoring

```
#include "G4RunManager.hh"
#include "G4ScoringManager.hh"

int main(int argc,char** argv)
{
 // Construct the run manager
 G4RunManager * runManager = new G4RunManager;

 // Activate command-based scorer
 G4ScoringManager::GetScoringManager();

 ...
}
```

### 4.9.2 Defining a scoring volume in the tracking world

Scoring volume can be declared as a logical volume that is already defined as a part of the mass geometry through /score/create/realWorldLogVol <LV_name> <anc_lvl> command, where <LV_name> is the name of G4LogicalVolume defined in the tracking world. If there are more than one physical volumes that share the same logical volume, scores are made for each individual physical volumes separately. Copy number of the physical volume is the index. If the physical volume is placed only once in its mother volume, but its (grand-)mother volume is duplicated, use the <anc_lvl> parameter to indicate the ancestor level where the copy number should be taken as the index of the score.

### 4.9.3 Defining a scoring mesh

To define a scoring mesh, the user has to specify the following.

- Shape and name of the 3D scoring mesh. Currently, box and cylinder are the only available shapes.
- Size of the scoring mesh. Mesh size must be specified as "half width" similar to the arguments of `G4Box` or `G4Tubs`, respectively .
- Number of bins for each axes. Note that too high number causes immense memory consumption.
- Optionally, position and rotation of the mesh. If not specified, the mesh is positioned at the center of the world volume without rotation.

The following sample UI commands define a scoring mesh named `boxMesh_1`, size of which is 2 m * 2 m * 2 m, and sliced into 30 cells along each axes.

Listing 4.29: UI commands to define a scoring mesh

```
#
# define scoring mesh
#
/score/create/boxMesh boxMesh_1
/score/mesh/boxSize 100. 100. 100. cm
/score/mesh/nBin 30 30 30
```

### 4.9.4 Defining a scoring probe

User may locate scoring "probes" at arbitrary locations. A "probe" is a virtual cube, the size of which has to be specfied as "half width". Given probes are located in an artificial "parallel world", probes may overlap to the volumes defined in the mass geometry, as long as probes themselves are not overlapping to each other or protruding from the world volume.

In addition, the user may optionally set a material to the probe. Once a material is set to the probe, it overwrites the material(s) defined in the tracking geometry when a track enters the probe cube. This material has to be already instantiated in user's detector construction class or defined in `G4NISTmanager`.

Because of this overwriting, physics quantities that depend on material or density, e.g. energy deposition or dose, would be measured according to the specified material. Please note that this overwriting material obviously affects to the simulation results, so the size and number of probes should be reasonably small to avoid significant side effects.

If probes are placed more than once, all probes have the same scorers but score individually.

The following sample UI commands define a scoring probe named `Probes`, size of which is 10 cm * 10 cm * 10 cm, filled by `G4_Water`, and located at three positions.

Listing 4.30: UI commands to define a scoring probe

```
#
# define scoring probe
#
/score/create/probe Probes 5. cm
/score/probe/material G4_WATER
/score/probe/locate 0. 0. 0. cm
/score/probe/locate 25. 0. 0. cm
/score/probe/locate 0. 25. 0. cm
```

### 4.9.5 Defining primitive scorers to a scoring volume

Once the scoring volume is defined, the user can define arbitrary scoring quantities and filters.

For a scoring volume the user may define arbitrary number primitive scorers to score for each physical volume (each cell for scoring mesh and each probe for scoring probe). For each scoring quantity, the use can set one filter. Please note that `/score/filter` commad affects on the immediately preceding scorer.

Names of scorers and filters must be unique for the scoring volume. It is possible to define more than one scorers of same kind with different names and, likely, with different filters. The list of available primitive scorers can be found at Table 4.1.

Defining a scoring volume and primitive scores should terminate with the `/score/close` command. The following sample UI commands define a scoring mesh named `boxMesh_1`, size of which is 2 m * 2 m * 2 m, and sliced into 30 cells along each axes. For each cell energy deposition, number of steps of gamma, number of steps of electron and number of steps of positron are scored.

Listing 4.31: UI commands to define a scoring mesh and scorers

```
#
# define scoring mesh
#
/score/create/boxMesh boxMesh_1
/score/mesh/boxSize 100. 100. 100. cm
/score/mesh/nBin 30 30 30
#
# define scorers and filters
#
/score/quantity/energyDeposit eDep
/score/quantity/nOfStep nOfStepGamma
/score/filter/particle gammaFilter gamma
/score/quantity/nOfStep nOfStepEMinus
/score/filter/particle eMinusFilter e-
/score/quantity/nOfStep nOfStepEPlus
/score/filter/particle ePlusFilter e+
#
/score/close
#
```

### 4.9.6 Drawing scores for a scoring mesh

Once scores assigned to a scoring mesh are filled, it is possible to visualize these scores. The score is drawn on top of the mass geometry with the current visualization settings.

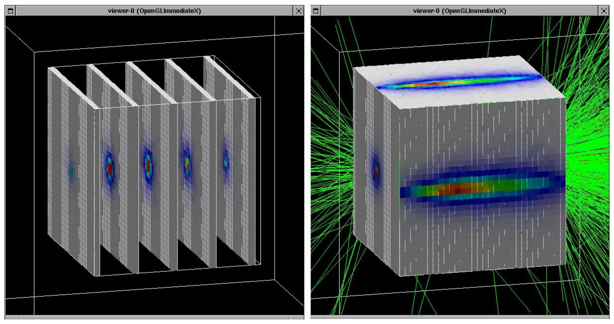


Fig. 4.9: Drawing scores in slices (left) and projection (right).

Scored data can be visualized using the commands `/score/drawProjection` and `/score/drawColumn`. For details, see examples/extended/runAndEvent/RE03.

By default, entries are linearly mapped to colors (gray - blue - green - red). This color mapping is implemented in `G4DefaultLinearColorMap` class, and registered to `G4ScoringManager` with the color map name `"defaultLinearColorMap"`. The user may alternate color map by implementing a customised color map class derived from `G4VScoreColorMap` and register it to `G4ScoringManager`. Then, for each `draw` command, one can specify the preferred color map.

This drawing funactionality is available only for scoring mesh.

### 4.9.7 Writing scores to a file

It is possible to dump a score in a mesh (`/score/dumpQuantityToFile` command) or all scores in a mesh (`/score/dumpAllQuantitiesToFile` command) to a file. The default file format is the simple CSV. To alternate the file format, one should overwrite `G4VScoreWriter` class and register it to `G4ScoringManager`. The scoring manager takes ownership of the registered writer, and will delete it at the end of the job.

Please refer to `/examples/extended/runAndEvent/RE03` for details.

### 4.9.8 Filling 1-D histogram

Through the template interface class `G4TScoreHistFiller` a primitive scorer can directly fill a 1-D histogram defined by `G4Analysis` module. Track-by-track or step-by-step filling allows command-based histogram such as energy spectrum. `G4TScoreHistFiller` template class must be instantiated in the user's code (e.g. in the constructor of user run action) with his/her choice of analysis data format.

Listing 4.32: Instantiation of `G4TScoreHistFiller`

```
#include "G4AnalysisManager.hh"
#include “G4TScoreHistFiller.hh”

auto histFiller = new G4TScoreHistFiller<G4AnalysisManager>;
```

`/score/fill1D <histID> <volName> <primName> <copNo>` command defines the histogram `<histID>` to be filled by `<primName>` primitive scorer assigned to `<volName>` scoring volume. If scoring volume in tracking world or probe is placed more than once, fill1D command should be issued for each individual copy number <copNo>.. Histogram <histID> must be defined through /analysis/h1/create command in advance to setting it to a primitive scorer. Scoring volume <volName> (either tracking world scorer or probe scorer) as well as the primitive scorer <primName> must be defined in advance, as well. This ifilling 1-D histogram functionality is not available for mesh scorer due to memory consumption concern. The list of primitive scorers available for 1-D histogram can be found at Table 4.1.

The following UI commands define a scoring probe named Probes, size of which is 10 cm * 10 cm * 10 cm, with two volume flux primitive scorers (one for total flux and the other for proton flux), and fill 1-D histograms of these two fluxes.

Listing 4.33: Filling 1-D histograms

```
#
# define scoring probe
#
/score/create/probe Probes 5. cm
/score/probe/locate 0. 0. 0. cm
#
# define flux scorers and filter
#
/score/quantity/volumeFlux volFlux
/score/quantity/volumeFlux protonFlux
/score/filter/particle protonFilter proton
/score/close
#
# define histograms
#
/analysis/h1/create volFlux Probes_volFlux 100 0.01 2000. MeV ! log
/analysis/h1/create protonFlux Probes_protonFlux 100 0.01 2000. MeV ! log
#
# filling histograms
#
/score/fill1D 1 Probes volFlux
/score/fill1D 2 Probes protonFlux
```

### 4.9.9 List of available primitive scorers

A primitive scorer is assigned to the scoring volume by /score/quantity/xxxxx <primName> <unit> where xxxxx is the name of primitive scorer listed below. Some of these primitive scorers can fill 1-D histogram described in the previous section.

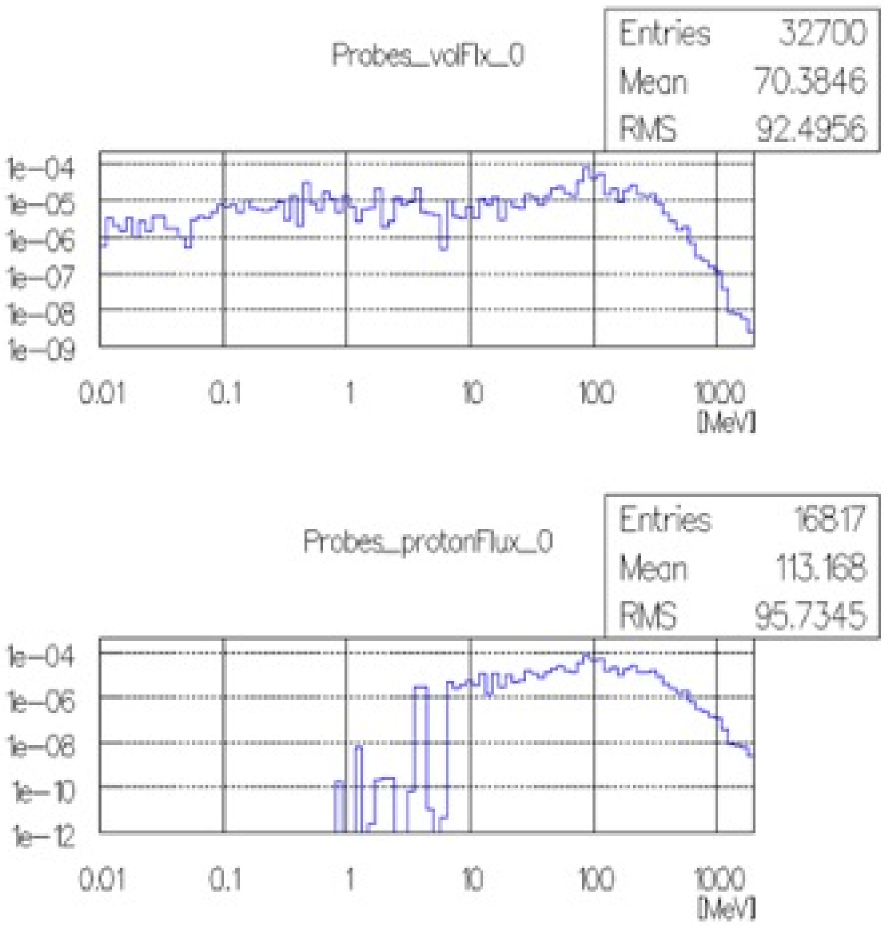


Fig. 4.10: Histograms of total flux (top) and proton flux (bottom)

Table 4.1: List of primitive scorers.

| Name of primitive scorer | Description | Default unit | x-axis of 1-D histogram | y-axis of 1-D histogram |
|---|---|---|---|---|
| cellCharge | deposited charge in the volume | e+ | n/a | n/a |
| cellFlux | sum of track length divided by the volume | $cm^{-2}$ | Ek in MeV | weighted cell flux |
| doseDeposit | deposited dose in the volume | Gy | dose per step in Gy | track weight |
| energyDeposit | deposited energy in the volume | MeV | eDep per step in MeV | track weight |
| flatSurfaceCurrent | surface current on -z surface to be used only for Box | $cm^{-2}$ | Ek in MeV | weighted current |
| flatSurfaceFlux | surface flux (1/cos(theta)) on -z surface to be used only for Box | $cm^{-2}$ | Ek in MeV | weighted flux |
| nOfCollision | number of steps made by physics interaction | n/a | n/a | n/a |
| nOfSecondary | number od secondary tracks generated in the volume | n/a | Ek in MeV | track weight |
| nOfStep | number of steps in the volume | n/a | step length in mm | entry (unweighted) |
| nOfTerminatedTrack | numver of tracks terminated in the volume (due to decay, interaction, stop, etc.) | n/a | n/a | n/a |
| nOfTrack | number of tracks in the volume (including both passing and terminated tracks) | n/a | Ek in MeV | track weight |
| passageCellCurrent | number of tracks that pass through the volume | n/a | Ek in MeV | track weight |
| passageCellFlux | sum of track length divided by the volume counted only for tracks that pass through the volume | $cm^{-2}$ | Ek in MeV | weighted cell flux |
| passageTrackLength | sum of track length in the volume for tracks that pass through the volume | mm | track length in mm | entry (unweighted) |
| population | number of tracks in the volume that are unique in an event | n/a | n/a | n/a |
| trackLength | total track length in the volume (including both passing and terminated tracks) | mm | n/a | n/a |
| volumeFlux | number of tracks getting into the volume | n/a | Ek in MeV | track weight |

CHAPTER

FIVE

# TRACKING AND PHYSICS

## 5.1 Tracking

### 5.1.1 Basic Concepts

**Philosophy of Tracking**

All GEANT4 processes, including the transportation of particles, are treated generically. In spite of the name "*tracking*", particles are not *transported* in the tracking category. `G4TrackingManager` is an interface class which brokers transactions between the event, track and tracking categories. An instance of this class handles the message passing between the upper hierarchical object, which is the event manager, and lower hierarchical objects in the tracking category. The event manager is a singleton instance of the `G4EventManager` class.

The tracking manager receives a track from the event manager and takes the actions required to finish tracking it. `G4TrackingManager` aggregates the pointers to `G4SteppingManager, G4Trajectory` and `G4UserTrackingAction`. Also there is a "use" relation to `G4Track` and `G4Step`.

`G4SteppingManager` plays an essential role in tracking the particle. It takes care of all message passing between objects in the different categories relevant to transporting a particle (for example, geometry and interactions in matter). Its public method `Stepping()` steers the stepping of the particle. The algorithm to handle one step is given below.

1. If the particle stop (i.e. zero kinetic energy), each active AtRest process proposes a step length in time based on the interaction it describes. And the process proposing the smallest step length will be invoked.
2. Each active discrete or continuous process must propose a step length based on the interaction it describes. The smallest of these step lengths is taken.
3. The geometry navigator calculates "Safety", the distance to the next volume boundary. If the minimum physical-step-length from the processes is shorter than "Safety", the physical-step-length is selected as the next step length. In this case, no further geometrical calculations will be performed.
4. If the minimum physical-step-length from the processes is longer than "Safety", the distance to the next boundary is re-calculated.
5. The smaller of the minimum physical-step-length and the geometric step length is taken.
6. All active continuous processes are invoked. Note that the particle's kinetic energy will be updated only after all invoked processes have completed. The change in kinetic energy will be the sum of the contributions from these processes.
7. The current track properties are updated before discrete processes are invoked. In the same time, the secondary particles created by processes are stored in SecondaryList. The updated properties are:
    - the kinetic energy of the current track particle (note that 'sumEnergyChange' is the sum of the energy difference before and after each process invocation)
    - position and time
8. The kinetic energy of the particle is checked to see whether or not it has been terminated by a continuous process. If the kinetic energy goes down to zero, AtRest processes will be applied at the next step if applicable.
9. The discrete process is invoked. After the invocation,
    - the energy, position and time of the current track particle are updated, and

- the secondaries are stored in SecondaryList.

10. The track is checked to see whether or not it has been terminated by the discrete process.
11. "Safety" is updated.
12. If the step was limited by the volume boundary, push the particle into the next volume.
13. Handle hit information.
14. Invoke the user intervention `G4UserSteppingAction`.
15. Save data to Trajectory.
16. Update the mean free paths of the discrete processes.
17. If the parent particle is still alive, reset the maximum interaction length of the discrete process which has occurred.
18. One step completed.

**What is a Process?**

Only processes can change information of `G4Track` and add secondary tracks via `ParticleChange`. `G4VProcess` is a base class of all processes and it has 3 kinds of `DoIt` and `GetPhysicalInteraction` methods in order to describe interactions generically. If a user want to modify information of `G4Track`, he (or she) SHOULD create a special process for the purpose and register the process to the particle.

**What is a Track?**

`G4Track` keeps 'current' information of the particle. (i.e. energy,momentum, position ,time and so on) and has 'static' information (i.e. mass, charge, life and so on) also. Note that `G4Track` keeps information at the beginning of the step while the `AlongStepDoIts` are being invoked for the step in progress.After finishing all `AlongStepDoIts`, `G4Track` is updated. On the other hand, `G4Track` is updated after each invocation of a `PostStepDoIt`.

**What is a Step?**

`G4Step` stores the transient information of a step. This includes the two endpoints of the step, `PreStepPoint` and `PostStepPoint`, which contain the points' coordinates and the volumes containing the points. `G4Step` also stores the change in track properties between the two points. These properties, such as energy and momentum, are updated as the various active processes are invoked.

**What is a ParticleChange?**

Processes do NOT change any information of `G4Track` directly in their `DoIt`. Instead, they proposes changes as a result of interactions by using `ParticleChange`. After each `DoIt`, `ParticleChange` updates `PostStepPoint` based on proposed changes. Then, `G4Track` is updated after finishing all `AlongStepDoIts` and after each `PostStepDoIt`.

## 5.1.2 Access to Track and Step Information

### How to Get Track Information

Track information may be accessed by invoking various `Get` methods provided in the `G4Track` class. For details, see the G4Track.hh header file in `$G4INCLUDE`. Typical information available includes:

- (x,y,z)
- Global time (time since the event was created)
- Local time (time since the track was created)
- Proper time (time in its rest frame since the track was created )
- Momentum direction ( unit vector )
- Kinetic energy
- Accumulated geometrical track length
- Accumulated true track length
- Pointer to dynamic particle
- Pointer to physical volume
- Track ID number

- Track ID number of the parent
- Current step number
- Track status
- (x,y,z) at the start point (vertex position) of the track
- Momentum direction at the start point (vertex position) of the track
- Kinetic energy at the start point (vertex position) of the track
- Pointer to the process which created the current track

**How to Get Step Information**

Step and step-point information can be retrieved by invoking various `Get` methods provided in the `G4Step`/`G4StepPoint` classes..

Information in G4Step includes:

- Pointers to `PreStep` and `PostStepPoint`
- Geometrical step length (step length before the correction of multiple scattering)
- True step length (step length after the correction of multiple scattering)
- Increment of position and time between `PreStepPoint` and `PostStepPoint`
- Increment of momentum and energy between `PreStepPoint` and `PostStepPoint`. (Note: to get the energy deposited in the step, you cannot use this 'Delta energy'. You have to use 'Total energy deposit' as below.)
- Pointer to `G4Track`
- Total energy deposited during the step - this is the sum of
  - the energy deposited by the energy loss process, and
  - the energy lost by secondaries which have NOT been generated because each of their energies was below the cut threshold
- Energy deposited not by ionization during the step
- Secondary tracks created during tracking for the current track.
  - NOTE: all secondaries are included. NOT only secondaries created in the CURRENT step.

Information in G4StepPoint (`PreStepPoint` and `PostStepPoint`) includes:

- (x, y, z, t)
- (px, py, pz, Ek)
- Momentum direction (unit vector)
- Pointers to physical volumes
- Safety
- Beta, gamma
- Polarization
- Step status
- Pointer to the physics process which defined the current step and its `DoIt` type
- Pointer to the physics process which defined the previous step and its `DoIt` type
- Total track length
- Global time (time since the current event began)
- Local time (time since the current track began)
- Proper time

**How to Get "particle change"**

Particle change information can be accessed by invoking various `Get` methods provided in the G4ParticleChange class. Typical information available includes:

- final momentum direction of the parent particle
- final kinetic energy of the parent particle
- final position of the parent particle
- final global time of the parent particle
- final proper time of the parent particle

- final polarization of the parent particle
- status of the parent particle (`G4TrackStatus`)
- true step length (this is used by multiple scattering to store the result of the transformation from the geometrical step length to the true step length)
- local energy deposited - this consists of either
  - energy deposited by the energy loss process, or
  - the energy lost by secondaries which have NOT been generated because each of their energies was below the cut threshold.
- number of secondaries particles
- list of secondary particles (list of `G4Track`)

### 5.1.3 Handling of Secondary Particles

Secondary particles are passed as `G4Tracks` from a physics process to tracking. `G4ParticleChange` provides the following four methods for a physics process:

- `AddSecondary( G4Track* aSecondary )`
- `AddSecondary( G4DynamicParticle* aSecondary )`
- `AddSecondary( G4DynamicParticle* aSecondary, G4ThreeVector position )`
- `AddSecondary( G4DynamicParticle* aSecondary, G4double time)`

In all but the first, the construction of `G4Track` is done in the methods using information given by the arguments.

### 5.1.4 User Actions

There are two classes which allow the user to intervene in the tracking. These are:

- `G4UserTrackingAction`, and
- `G4UserSteppingAction`.

Each provides methods which allow the user access to the GEANT4 kernel at specific points in the tracking.

**Note:** Users SHOULD NOT (and CAN NOT) change `G4Track` in `UserSteppingAction`. The only exception is the `TrackStatus`.

**Note:** Users have to be cautious to implement an unnatural/unphysical action in these user actions. See the section *Killing Tracks in User Actions and Energy Conservation* for more details.

### 5.1.5 Verbose Outputs

The verbose information output flag can be turned on or off. The amount of information printed about the track/step, from brief to very detailed, can be controlled by the value of the verbose flag, for example,

```
G4UImanager* UI = G4UImanager::GetUIpointer();

UI->ApplyCommand("/tracking/verbose 1");
```

### 5.1.6 Trajectory and Trajectory Point

#### G4Trajectory and G4TrajectoryPoint

`G4Trajectory` and `G4TrajectoryPoint` are default concrete classes provided by GEANT4, which are derived from the `G4VTrajectory` and `G4VTrajectoryPoint` base classes, respectively. A `G4Trajectory` class object is created by `G4TrackingManager` when a `G4Track` is passed from the `G4EventManager`. `G4Trajectory` has the following data members:

- ID numbers of the track and the track's parent
- particle name, charge, and PDG code
- a collection of `G4TrajectoryPoint` pointers

`G4TrajectoryPoint` corresponds to a step point along the path followed by the track. Its position is given by a `G4ThreeVector`. A `G4TrajectoryPoint` class object is created in the `AppendStep()` method of `G4Trajectory` and this method is invoked by `G4TrackingManager` at the end of each step. The first point is created when the `G4Trajectory` is created, thus the first point is the original vertex.

The creation of a trajectory can be controlled by invoking `G4TrackingManager::SetStoreTrajectory(G4bool)`. The UI command */tracking/storeTrajectory _bool_* does the same. The user can set this flag for each individual track from his/her `G4UserTrackingAction::PreUserTrackingAction()` method.

**Note:** The user should not create trajectories for secondaries in a shower due to the large amount of memory consumed.

All the created trajectories in an event are stored in `G4TrajectoryContainer` class object and this object will be kept by `G4Event`. To draw or print trajectories generated in an event, the user may invoke the `DrawTrajectory()` or `ShowTrajectory()` methods of `G4VTrajectory`, respectively, from his/her `G4UserEventAction::EndOfEventAction()`. The geometry must be drawn before the trajectory drawing. The color of the drawn trajectory depends on the particle charge:

- negative: red
- neutral: green
- positive: blue

**Note:** Due to improvements in `G4Navigator`, a track can execute more than one turn of its spiral trajectory without being broken into smaller steps as long as the trajectory does not cross a geometrical boundary. Thus a drawn trajectory may not be circular.

#### Customizing trajectory and trajectory point

`G4Track` and `G4Step` are transient classes; they are not available at the end of the event. Thus, the concrete classes `G4VTrajectory` and `G4VTrajectoryPoint` are the only ones a user may employ for end-of-event analysis or for persistency. As mentioned above, the default classes which GEANT4 provides, i.e. `G4Trajectory` and `G4TrajectoryPoint`, have only very primitive quantities. The user can customize his/her own trajectory and trajectory point classes by deriving directly from the respective base classes.

To use the customized trajectory, the user must construct a concrete trajectory class object in the `G4UserTrackingAction::PreUserTrackingAction()` method and make its pointer available to `G4TrackingManager` by using the `SetTrajectory()` method. The customized trajectory point class object must be constructed in the `AppendStep()` method of the user's implementation of the trajectory class. This `AppendStep()` method will be invoked by `G4TrackingManager`.

To customize trajectory drawing, the user can override the `DrawTrajectory()` method in his/her own trajectory class.

When a customized version of G4Trajectory declares any new class variables, *operator new* and *operator delete* must be provided. It is also useful to check that the allocation size in *operator new* is equal to `sizeof(G4Trajectory)`. These two points do not apply to `G4VTrajectory` because it has no *operator new* or *operator delete*.

## 5.2 Physics Processes

### 5.2.1 Overview

Physics processes describe how particles interact with a material. Seven major categories of processes are provided by GEANT4:

1. electromagnetic,
2. hadronic,
3. decay,
4. photolepton-hadron,
5. optical,
6. parameterization, and
7. transportation.

The generalization and abstraction of physics processes is a key issue in the design of GEANT4. All physics processes are treated in the same manner from the tracking point of view. The GEANT4 approach enables anyone to create a process and assign it to a particle type. This openness should allow the creation of processes for novel, domain-specific or customised purposes by individuals or groups of users.

Each process has two groups of methods which play an important role in tracking, `GetPhysicalInteractionLength` (GPIL) and `DoIt`. The GPIL method gives the step length from the current space-time point to the next space-time point. It does this by calculating the probability of interaction based on the process's cross section information. At the end of this step the `DoIt` method should be invoked. The `DoIt` method implements the details of the interaction, changing the particle's energy, momentum, direction and position, and producing secondary tracks if required. These changes are recorded as `G4VParticleChange` objects (see *Particle change*).

#### G4VProcess

`G4VProcess` is the base class for all physics processes. Each physics process must implement virtual methods of `G4VProcess` which describe the interaction (DoIt) and determine when an interaction should occur (GPIL). In order to accommodate various types of interactions `G4VProcess` provides three `DoIt` methods:

- `G4VParticleChange* AlongStepDoIt( const G4Track& track, const G4Step& stepData )`
  This method is invoked while `G4SteppingManager` is transporting a particle through one step. The corresponding `AlongStepDoIt` for each defined process is applied for every step regardless of which process produces the minimum step length. Each resulting change to the track information is recorded and accumulated in `G4Step`. After all processes have been invoked, changes due to `AlongStepDoIt` are applied to `G4Track`, including the particle relocation and the safety update. Note that after the invocation of `AlongStepDoIt`, the endpoint of the `G4Track` object is in a new volume if the step was limited by a geometric boundary. In order to obtain information about both the old and new volumes, `G4Step` must be accessed, since it contains information about both pre-step and post-step points of a step.
- `G4VParticleChange* PostStepDoIt( const G4Track& track, const G4Step& stepData )`

This method is invoked at the end point of a step, only if its process limit the step, or if the process is forced to occur at each step. `G4Track` will be updated after each invocation of `PostStepDoIt`.
- `G4VParticleChange* AtRestDoIt( const G4Track& track, const G4Step& stepData )`
This method is invoked only for stopped particles, and only if its process limit the step in time, or if the process is forced to occur.

For each of the above `DoIt` methods `G4VProcess` provides a corresponding pure virtual GPIL method:

- `PostStepGetPhysicalInteractionLength`

```
G4double PostStepGetPhysicalInteractionLength(const G4Track& track,
                                              G4double previousStepSize,
                                              G4ForceCondition* condition )
```

  This method generates the step length allowed by its process. It also provides a flag to force the interaction to occur regardless of its step length.
- `AlongStepGetPhysicalInteractionLength`

```
G4double AlongStepGetPhysicalInteractionLength(const G4Track& track,
                                               G4double previousStepSize,
                                               G4double currentMinimumStep,
                                               G4double& proposedSafety,
                                               G4GPILSelection* selection )
```

  This method generates the step length allowed by its process.
- `AtRestGetPhysicalInteractionLength`

```
G4double AtRestGetPhysicalInteractionLength(const G4Track& track,
                                            G4ForceCondition* condition )
```

  This method generates the step length in time proposed by this process. It also provides a flag to force the interaction to occur regardless of its step length.

Other pure virtual methods in `G4VProcess` follow:

- `virtual G4bool IsApplicable(const G4ParticleDefinition&)`
returns true if this process object is applicable to the particle type.
- `virtual void PreparePhysicsTable(const G4ParticleDefinition&)` and
- `virtual void BuildPhysicsTable(const G4ParticleDefinition&)`
is messaged by the process manager, whenever cross section tables should be prepared and rebuilt due to changing cut-off values. It is not mandatory if the process is not affected by cut-off values.
- `virtual void StartTracking()` and
- `virtual void EndTracking()`
are messaged by the tracking manager at the beginning and end of tracking the current track.
- `virtual const G4VProcess* GetCreatorProcess() const`
returns the sub-process pointer to be used as CreatorProcess for secondaries produced at the given step. It is needed for combined processes like G4GammaGeneralProcess or G4NeutronGeneralProcess. Other methods:
- `virtual const G4String& GetProcessName() const`
- `virtual G4ProcessType GetProcessType() const`
- `virtual G4int GetProcessSubType() const`
are useful for control on MC truth in an application and debugging.

### Other base classes for processes

Specialized processes may be derived from seven additional virtual base classes which are themselves derived from `G4VProcess`. Three of these classes are used for simple processes:

**`G4VRestProcess`** Processes using only the `AtRestDoIt` method.
example: neutron capture

**`G4VDiscreteProcess`** Processes using only the `PostStepDoIt` method.
example: Compton scattering, hadron inelastic interaction. There are virtual methods, which are needed for more accurate tracking of charged particles:

- `virtual G4double GetCrossSection(const G4double kinE, const G4MaterialCutsCouple*)`
- `virtual G4double MinPrimaryEnergy(const G4ParticleDefinition*, const G4Material*)`

The other four classes are provided for rather complex processes:

**`G4VContinuousDiscreteProcess`** Processes using both `AlongStepDoIt` and `PostStepDoIt` methods.
example: transportation, ionisation(energy loss and delta ray)

**`G4VRestDiscreteProcess`** Processes using both `AtRestDoIt` and `PostStepDoIt` methods.
example: positron annihilation, decay (both in flight and at rest)

**`G4VRestContinuousProcess`** Processes using both `AtRestDoIt` and `AlongStepDoIt` methods.

**`G4VRestContinuousDiscreteProcess`** Processes using `AtRestDoIt`, `AlongStepDoIt and` PostStep-DoIt methods.

### Particle change

`G4VParticleChange` and its descendants are used to store the final state information of the track, including secondary tracks, which has been generated by the `DoIt` methods. The instance of `G4VParticleChange` is the only object whose information is updated by the physics processes, hence it is responsible for updating the step. The stepping manager collects secondary tracks and only sends requests via particle change to update `G4Step`.

`G4VParticleChange` is introduced as an abstract class. It has a minimal set of methods for updating `G4Step` and handling secondaries. A physics process can therefore define its own particle change derived from `G4VParticleChange`. Three pure virtual methods are provided,

- `virtual G4Step* UpdateStepForAtRest( G4Step* step)`,
- `virtual G4Step* UpdateStepForAlongStep( G4Step* step )`, and
- `virtual G4Step* UpdateStepForPostStep( G4Step* step)`,

which correspond to the three `DoIt` methods of `G4VProcess`. Each derived class should implement these methods. There are specialized derived classes

- `G4ParticleChange` - used in hadronic physics processes,
- `G4ParticleChangeForTransport` - used for transportation processes,
- `G4ParticleChangeForDecay` - used for `G4Decay` and `G4RadioactiveDecay`,
- `G4ParticleChangeForLoss` - used for energy loss processes,
- `G4ParticleChangeForMSC` - used for multiple scattering processes,
- `G4ParticleChangeForGamma` - used for discrete electromagnetic processes.

### 5.2.2 Electromagnetic Interactions

This section summarizes the electromagnetic (EM) physics processes which are provided with GEANT4. Extended information are available at EM web pages. For details on the implementation of these processes please refer to the Physics Reference Manual.

To use the electromagnetic physics data files are needed. The user should set the environment variable G4LEDATA to the directory with this files. These files are distributed together with GEANT4 and can be obtained via the GEANT4 download web page.

#### Electromagnetic Processes

The following is a summary of the electromagnetic processes available in GEANT4.

- Photon processes
  - Gamma conversion (also called pair production, class name `G4GammaConversion`)
  - Photo-electric effect (class name `G4PhotoElectricEffect`)
  - Compton scattering (class name `G4ComptonScattering`)
  - Rayleigh scattering (class name `G4RayleighScattering`)
  - Muon pair production (class name `G4GammaConversionToMuons`)
  - X-ray reflection (class name `G4XrayReflection`)
  - General gamma process (class name `G4GeneralGammaProcess`)
- Electron/positron processes
  - Ionisation and delta ray production (class name `G4eIonisation`)
  - Bremsstrahlung (class name `G4eBremsstrahlung`)
  - e+e- pair production (class name `G4ePairProduction`)
  - Multiple scattering (class name `G4eMultipleScattering`)
  - Positron annihilation into two gammas (class name `G4eplusAnnihilation`)
  - Positron annihilation into two muons (class name `G4AnnihiToMuPair`)
  - Positron annihilation into hadrons (class name `G4eeToHadrons`)
  - Combined process for multiple scattering and transportation (class name `G4TransportationWithMsc`)
- Muon processes
  - Ionisation and delta ray production (class name `G4MuIonisation`)
  - Bremsstrahlung (class name `G4MuBremsstrahlung`)
  - e+e- pair production (class name `G4MuPairProduction`)
  - mu+mu- pair production (class name `G4MuonToMuonPairProduction`)
  - Multiple scattering (class name `G4MuMultipleScattering`)
- Hadron/ion processes
  - Ionisation (class name `G4hIonisation`)
  - Ionisation for ions (class name `G4ionIonisation`)
  - Ionisation for heavy exotic particles (class name `G4hhIonisation`)
  - Ionisation for classical magnetic monopole (class name `G4mplIonisation`)
  - Multiple scattering (class name `G4hMultipleScattering`)
  - Bremsstrahlung (class name `G4hBremsstrahlung`)
  - e+e- pair production (class name `G4hPairProduction`)
- Coulomb scattering processes
  - Alternative process for simulation of single Coulomb scattering of all charged particles (class name `G4CoulombScattering`)
  - Alternative process for simulation of single Coulomb scattering of ions (class name `G4ScreenedNuclearRecoil`)
- Processes for simulation of polarized electron and gamma beams
  - Compton scattering of circularly polarized gamma beam on polarized target (class name `G4PolarizedCompton`)

  - Pair production induced by circularly polarized gamma beam (class name `G4PolarizedGammaConversion`)
  - Photo-electric effect induced by circularly polarized gamma beam (class name `G4PolarizedPhotoElectricEffect`)
  - Bremsstrahlung of polarized electrons and positrons (class name `G4ePolarizedBremsstrahlung`)
  - Ionisation of polarized electron and positron beam (class name `G4ePolarizedIonisation`)
  - Annihilation of polarized positrons (class name `G4eplusPolarizedAnnihilation`)
- Processes for simulation of X-rays and optical protons production by charged particles
  - Synchrotron radiation (class name `G4SynchrotronRadiation`)
  - Transition radiation (class name `G4TransitionRadiation`)
  - Cerenkov radiation (class name `G4Cerenkov`)
  - Scintillations (class name `G4Scintillation`)
- The processes described above use physics model classes, which may be combined according to particle energy. It is possible to change the energy range over which different models are valid, and to apply other models specific to particle type, energy range, and G4Region. Models, which are used in the default EM Physics List, are mainly from the standard EM sub-library:
  - Photoelectric effect (class name `G4LivermorePhotoElectricModel`)
  - Compton scattering (class name `G4KleinNishinaCompton`)
  - Electron/positron pair production (class name `G4PairProductionRelModel`)
  - Rayleigh scattering (class name `G4LivermoreRayleighModel`)
  - Multiple scattering (class name `G4UrbanMscModel`)
  - Multiple scattering (class name `G4WentzelVIModel`)
  - Single Coulomb scattering (class name `G4eCoulombScatteringModel`)
  - Electron ionisation (class name `G4MollerBhabhaModel`)
  - Electron/positron bremsstrahlung (class name `G4SeltzerBergerModel`)
  - Electron/positron bremsstrahlung (class name `G4eBremsstrahlungRelModel`)
  - Positron annihilation into 2 gamma (class name `G4eeToTwoGammaModel`)
  - Muon and hadron low-energy ionisation (class name `G4BraggModel`)
  - Ion low-energy ionisation (class name `G4BraggIonModel`)
  - Ionisation of ions (class name `G4LindhardSorensenIonModel`)
  - Anti-particle low-energy ionisation (class name `G4ICRU73QOModel`)
  - Muon and hadron ionisation (class name `G4BetheBlochModel`)
  - Muon ionisation (class name `G4MuBetheBlochModel`)
  - Muon bremsstrahlung (class name `G4MuBremsstrahlungModel`)
  - Muon pair production by muons (class name `G4MuonToMuonPairProductionModel`)
  - Hadron bremsstrahlung (class name `G4hBremsstrahlungModel`)
  - Muon e+e- pair production (class name `G4MuPairProductionModel`)
  - Hadron e+e- pair production (class name `G4hPairProductionModel`)

  The following alternative models are available in the standard EM sub-library:
  - Ionisation in thin absorbers (class name `G4PAIModel`)
  - Ionisation in thin absorbers (class name `G4PAIPhotModel`)
  - Ionisation in low-density media (class name `G4BraggIonGasModel`)
  - Ionisation in low-density media (class name `G4BetheBlochIonGasModel`)
  - Ionisation of relativistic ions (class name `G4AtimaEnergyLossModel`)
  - Electron/positron pair production (class name `G4BetheHeitlerModel`)
  - Electron/positron pair production (class name `G4BetheHeitler5DModel`)
  - Positron annihilation into 2 or 3 gamma (class name `G4eplusTo2GammaOKVIModel`)
  - Multiple scattering (class name `G4GoudsmitSaundersonMscModel`)
  - Multiple scattering (class name `G4LowEWentzelVIModel`)
  - Single scattering (class name `G4eSingleCoulombScatteringModel`)
  - Single scattering (class name `G4hCoulombScatteringModel`)
  - Single scattering (class name `G4IonCoulombScatteringModel`)

  In the low-energy sub-library there are alternative models (more detailes see below):
  - Photoelectric effect (class name `G4PenelopePhotoElectricModel`)

  - Compton scattering (class name `G4PenelopeComptonModel`)
  - Compton scattering (class name `G4LivermoreComptonModel`)
  - Compton scattering (class name `G4LivermorePolarizedComptonModel`)
  - Compton scattering (class name `G4LowEPComptonModel`)
  - Compton scattering (class name `G4LowEPPolarizedComptonModel`)
  - Gamma conversion to e+e- pair (class name `G4LivermoreGammaConversionModel`)
  - Gamma conversion to e+e- pair (class name `G4LivermoreGammaConversion5DModel`)
  - Gamma conversion to e+e- pair (class name `G4PenelopeGammaConversionModel`)
  - Rayleigh scattering (class name `G4JAEAElasticScatteringModel`)
  - Rayleigh scattering (class name `G4JAEAPolarizedElasticScatteringModel`)
  - Rayleigh scattering (class name `G4LivermorePolarizedRayleighModel`)
  - Rayleigh scattering (class name `G4PenelopeRayleighModel`)
  - Electron ionisation (class name `G4LivermoreIonisationModel`)
  - Electron and positron ionisation (class name `G4PositronIonisationModel`)
  - Ion ionisation (class name `G4IonParametrisedLossModel`)
  - Electron, proton, alpha, and ion ionisation (class name `G4MicroElecInelastic`)
  - Electron, proton, alpha, and ion elastic scattering (class name `G4MicroElecElastic`)
  - Electron, proton, alpha, and ion ionisation (class name `G4MicroElecInelastic_new`)
  - Electron, proton, alpha, and ion elastic scattering (class name `G4MicroElecElastic_new`)

It is recommended to use physics constructor classes provided with reference physics lists (in subdirectory `source/physics_lists/constructors/electromagnetic` of the GEANT4 source distribution):

- default EM physics, multiple scattering is simulated with "UseSafety" type of step limitation by combined `G4WentzelVIModel` and `G4eCoulombScatteringModel` for all particle types, for of e+- below 100 MeV `G4UrbanMscModel` is used, `RangeFactor = 0.04`, `G4LivermorePhotoElectricModel` is used for simulation of the photo-electric effect, the Rayleigh scattering process is enabled below 1 MeV, `G4GammaGeneralProcess` is enabled, physics tables are built from 100 eV to 100 TeV, 7 bins per energy decade of physics tables are used (class name `G4EmStandardPhysics`),
- optional EM physics providing fast but less accurate electron transport due to "Minimal" method of step limitation by multiple scattering, `RangeFactor = 0.2`, Rayleigh scattering is disabled, photo-electric effect is using `G4PEEffectFluoModel`, enabled "ApplyCuts" option, and enabled `G4TransportationWithMsc` combined process (class name `G4EmStandardPhysics_option1`),
- optional EM physics providing fast but less accurate electron transport due to "Minimal" method of step limitation by multiple scattering, `RangeFactor = 0.2`, "Simple" method of step limitation by multiple scattering, Rayleigh scattering is disabled, and photo-electric effect is using `G4PEEffectFluoModel` (class name `G4EmStandardPhysics_option2`)
- EM physics for simulation with high accuracy due to "UseDistanceToBoundary" multiple scattering step limitation and usage of `G4UrbanMscModel` for all charged particles, `RangeFactor = 0.03`, reduced *finalRange* parameter of stepping function optimized per particle type, alternative model `G4KleinNishinaModel` for Compton scattering, enabled fluorescence, enabled nuclear stopping, `G4Generator2BS` angular generator for bremsstrahlung, `G4LindhardSorensenIonModel` for ion ionisation, `G4ePairProduction` for electron/positron, 20 bins energy decade of physics tables, and 10 eV low-energy limit for tables (class name `G4EmStandardPhysics_option3`)
- Combination of EM models for simulation with high accuracy includes multiple scattering with "UseSafetyPlus" type of step limitation by combined `G4WentzelVIModel` and `G4eCoulombScatteringModel` for all particle types, for of e+- below 100 MeV `G4GoudsmitSaundersonMscModel` is used, `RangeFactor = 0.08`, `Scin = 3` (error free stepping near geometry boundaries), reduced *finalRange* parameter of stepping function optimized per particle type, enabled fluorescence, enabled nuclear stopping, enable accurate angular generator for ionisation models, `G4LowEPComptonModel` below 20 MeV and `G4KleinNishinaModel` above, `G4BetheHeitler5DModel` for gamma conversion, `G4PenelopeIonisationModel` for electrons and positrons below 100 keV, `G4LindhardSorensenIonModel` for ion ionisation, `G4Generator2BS` angular generator for bremsstrahlung, `G4ePairProduction` for electron/positron, and 20 bins per energy decade of physics tables, (class name `G4EmStandardPhysics_option4`)
- Models based on Livermore data bases for electrons and gamma, enabled Rayleigh scattering, enabled fluores-

cence, `G4BetheHeitler5DModel` is used for gamma conversion, enabled nuclear stopping, enable accurate angular generator for ionisation models, `G4IonParameterisedLossModel` for ion ionisation, for of e+- below 100 MeV `G4GoudsmitSaundersonMscModel` is used with "UseSafetyPlus" multiple scattering step limitation, `RangeFactor = 0.08`, `Scin = 3` (error free stepping near geometry boundaries), `G4Generator2BS` angular generator for bremsstrahlung, `G4ePairProduction` for electron/positron, `G4LindhardSorensenIonModel` for ion ionisation, and 20 bins per energy decade of physics tables, (`G4EmLivermorePhysics`);

- Models based on Livermore data bases and new model for Compton scattering `G4LowEPComptonModel`, `G4BetheHeitler5DModel` is used for gamma conversion, low-energy model of multiple scattering `G4LowEWenzelMscModel`, and `G4LindhardSorensenIonModel`, `G4hBremsstrahlung` and `G4hPairProduction` for ions (`G4EmLowEPPhysics`);
- Penelope2008 models for electrons, positrons and gamma, enabled Rayleigh scattering, enabled fluorescence, enabled nuclear stopping, enable accurate angular generator for ionisation models, `G4LindhardSorensenIonModel` for ion ionisation, for of e+- below 100 MeV `G4GoudsmitSaundersonMscModel` is used with "UseSafetyPlus" multiple scattering step limitation, `RangeFactor = 0.08`, `Scin = 3` (error free stepping near geometry boundaries), and 20 bins per energy decade of physics tables, (`G4EmPenelopePhysics`);
- Experimental physics with multiple scattering of e+- below 100 MeV simulated by `G4GoudsmitSaundersonMscModel` is done on top of the default EM physics (`G4EmStandardPhysicsGS`);
- Experimental physics is done on top of the default EM physics with multiple scattering of e+- below 100 MeV simulated by a combination of `G4UrbanMscModel` below 1 MeV and `G4WentzelVIModel`, `G4eCoulombScatteringModel`, and for ions `G4LindhardSorensenIonModel` (`G4EmStandardPhysicsWVI`);
- Experimental physics with single scattering models instead of multiple scattering is done on top of the default EM physics, for all leptons and hadrons `G4eCoulombScatteringModel` is used, for ions - `G4IonCoulombScatteringModel` (`G4EmStandardPhysicsSS`);
- Low-energy GEANT4-DNA physics (`G4EmDNAPhysics`).
- Alternative low-energy GEANT4-DNA physics constructors (`G4EmDNAPhysics_optionX`, where X is 1 to 8). Refer to GEANT4-DNA section. The default upper energy applicability limit is 300 MeV. For particles or processes where DNA physics is not available the standard models are used.

Examples of the registration of these physics constructor and construction of alternative combinations of options are shown in basic, extended and advanced examples, which can be found in the subdirectories `examples/basic`, `examples/extended/electromagnetic`, `examples/medical`, `examples/advanced`, and of the GEANT4 source distribution. Examples illustrating the use of electromagnetic processes are available as part of the GEANT4 release.

**Options** are available for steering of electromagnetic processes. These options may be invoked either by UI commands or by the new C++ interface class `G4EmParameters`. The interface `G4EmParameters::Instance()` is thread safe, EM parameters are shared between threads, and parameters are shared between all EM processes. Parameters may be modified at G4State_PreInit or G4State_Idle states of GEANT4. Note, that when any of EM physics constructor is instantiated a default set of EM parameters for this EM physics configuration is defined. So, parameters modification should be applied only after. This class has the following public methods:

- Dump()
- StreamInfo(std::ostream&)
- SetDefaults()
- SetLossFluctuations(G4bool)
- SetBuildCSDARange(G4bool)
- SetLPM(G4bool)
- SetUseCutAsFinalRange(G4bool)
- SetApplyCuts(G4bool)
- SetFluo(G4bool val)
- SetFluoDirectory(G4EmFluoDirectory type)

- SetAuger(G4bool val)
- SetPixe(G4bool val)
- SetDeexcitationIgnoreCut(G4bool val)
- SetLateralDisplacement(G4bool val)
- SetLateralDisplacementAlg96(G4bool val)
- SetMuHadLateralDisplacement(G4bool val)
- ActivateAngularGeneratorForIonisation(G4bool val)
- SetUseMottCorrection(G4bool val)
- SetIntegral(G4bool val)
- SetBirksActive(G4bool val)
- SetUseICRU90Data(G4bool val)
- SetFluctuationType(G4EmFluctuationType type)
- SetDNAFast(G4bool val)
- SetDNAStationary(G4bool val)
- SetDNAElectronMsc(G4bool val)
- SetGeneralProcessActive(G4bool val)
- SetEnableSamplingTable(G4bool val)
- SetEnablePolarisation(G4bool val)
- SetDirectionalSplitting(G4bool val)
- SetQuantumEntanglement(G4bool val)
- SetRetrieveMuDataFromFile(G4bool val)
- SetPhotoeffectBelowKShell(G4bool val)
- SetMscPositronCorrection(G4bool v)
- SetOnIsolated(G4bool val)
- ActivateDNA(G4bool val)
- SetIsPrintedFlag(G4bool val);
- SetMinEnergy(G4double)
- SetMaxEnergy(G4double)
- SetMaxEnergyForCSDARange(G4double)
- SetLowestElectronEnergy(G4double)
- SetLowestMuHadEnergy(G4double)
- SetLowestTripletEnergy(G4double)
- SetLinearLossLimit(G4double)
- SetBremsstrahlungTh(G4double)
- SetMuHadBremsstrahlungTh(G4double val)
- SetLambdaFactor(G4double)
- SetFactorForAngleLimit(G4double)
- SetMscThetaLimit(G4double)
- SetMscEnergyLimit(G4double)
- SetMscRangeFactor(G4double)
- SetMscMuHadRangeFactor(G4double)
- SetMscGeomFactor(G4double)
- SetMscSafetyFactor(G4double)
- SetMscLambdaLimit(G4double)
- SetMscSkin(G4double)
- SetScreeningFactor(G4double)
- SetMaxNIELEnergy(G4double)
- SetMaxEnergyFor5DMuPair(G4double)
- SetStepFunction(G4double, G4double)
- SetStepFunctionMuHad(G4double, G4double)
- SetStepFunctionLightIons(G4double, G4double);
- SetStepFunctionIons(G4double, G4double);
- SetDirectionalSplittingRadius(G4double)
- SetDirectionalSplittingTarget(const G4ThreeVector&)

- SetNumberOfBinsPerDecade(G4int)
- SetVerbose(G4int)
- SetWorkerVerbose(G4int)
- SetTransportationWithMsc(G4TransportationWithMscType type)
- SetMscStepLimitType(G4MscStepLimitType val)
- SetMscMuHadStepLimitType(G4MscStepLimitType val)
- SetSingleScatteringType(G4eSingleScatteringType val)
- SetNuclearFormFactorType(G4NuclearFormFactorType val)
- SetDNAeSolvationSubType(G4DNAModelSubType val)
- SetConversionType(G4int val)
- SetPIXECrossSectionModel(const G4String&)
- SetPIXEElectronCrossSectionModel(const G4String&)
- SetLivermoreDataDir(const G4String&)
- AddPAIModel(const G4String& particle, const G4String& region, const G4String& type)
- AddMicroElec(const G4String& region)
- AddDNA(const G4String& region, const G4String& type)
- AddPhysics(const G4String& region, const G4String& physics_type)
- SetSubCutRegion(const G4String& region)
- SetDeexActiveRegion(const G4String& region, G4bool, G4bool, G4bool)
- SetProcessBiasingFactor(const G4String& process, G4double, G4bool)
- ActivateForcedInteraction(const G4String& process, const G4String& region, G4double, G4bool)
- ActivateSecondaryBiasing(const G4String& process, const G4String& region, G4double, G4double)
- SetEmSaturation(G4EmSaturation*)

The corresponding UI command can be accessed in the UI subdirectories "/process/eLoss", "/process/em", and "/process/msc". The following types of step limitation by multiple scattering are available:

- fMinimal - simplified step limitation (used in _EMV and _EMX Physics Lists)
- fUseSafety - default
- fUseDistanceToBoundary - advance method of step limitation used in EM examples, required parameter *skin* > *0* , should be used for setup without magnetic field
- fUseSafetyPlus - advanced method may be used with magnetic field

**G4EmCalculator** is a class which provides access to cross sections and stopping powers. This class can be used anywhere in the user code provided the physics list has already been initialised (G4State_Idle). `G4EmCalculator` has "Get" methods which can be applied to materials for which physics tables are already built, and "Compute" methods which can be applied to any material defined in the application or existing in the GEANT4 internal database. The public methods of this class are:

- GetDEDX(kinEnergy,particle,material,const G4Region* r=nullptr)
- GetRangeFromRestrictedDEDX(kinEnergy,particle,material,const G4Region* r=nullptr)
- GetCSDARange(kinEnergy,particle,material,const G4Region* r=nullptr)
- GetRange(kinEnergy,particle,material,const G4Region* r=nullptr)
- GetKinEnergy(range,particle,material,const G4Region* r=nullptr)
- GetCrossSectionPerVolume(kinEnergy,particle,material,const G4Region* r=nullptr)
- GetShellIonisationCrossSectionPerAtom(particle,Z,shell,kinEnergy)
- GetMeanFreePath(kinEnergy,particle,material,const G4Region* r=nullptr)
- PrintDEDXTable(particle)
- PrintRangeTable(particle)
- PrintInverseRangeTable(particle)
- ComputeDEDX(kinEnergy,particle,process,material,cut=DBL_MAX)
- ComputeElectronicDEDX(kinEnergy,particle,material,cut=DBL_MAX)
- ComputeDEDXForCutInRange(kinEnergy,particle,material,cut=DBL_MAX)
- ComputeNuclearDEDX(kinEnergy,particle,material,cut=DBL_MAX)
- ComputeTotalDEDX(kinEnergy,particle,material,cut=DBL_MAX)
- ComputeCrossSectionPerVolume(kinEnergy,particle,process,material,cut=nullptr)

- ComputeCrossSectionPerAtom(kinEnergy,particle,process,Z,A,cut=nullptr)
- ComputeCrossSectionPerShell(kinEnergy,particle,process,Z,shellIdx,cut=nullptr)
- ComputeGammaAttenuationLength(kinEnergy,material)
- ComputeShellIonisationCrossSectionPerAtom(particle,Z,shell,kinEnergy)
- ComputeMeanFreePath(kinEnergy,particle,process,material,cut=nullptr)
- ComputeEnergyCutFromRangeCut(range,particle,material)
- FindParticle(const G4String&)
- FindIon(G4int Z, G4int A)
- FindMaterial(const G4String&)
- FindRegion(const G4String&)
- FindCouple(const G4Material*, const const G4Region* r=nullptr)
- FindProcess(particle, const G4String& processName)
- SetVerbose(G4int)

For these interfaces, particles, materials, or processes may be pointers (`const G4ParticleDefinition*`, `const G4Material*`) or strings with names (`const G4String&`).

**G4NIELCalculator** is a class which provides computation of NIEL energy loss at a step independently on cuts and tracking. `G4NIELCalculator` has follow methods:

- ComputeNIEL(const G4Step*)
- RecoilEnergy(const G4Step*)
- AddEmModel(G4VEmModel*)

the last method allows customisation of NIEL model.

### Low Energy Electromagnetic Library

A physical interaction is described by a process class which can handle physics models, described by model classes. The following is a summary of the Low Energy Electromagnetic physics models available in GEANT4. Further information is available in the web pages of the GEANT4 Low Energy Electromagnetic Physics Working Group, accessible from the GEANT4 web site, "who we are" section, then "working groups".

The physics content of these models is documented in the GEANT4 Physics Reference Manual. They are based on the Livermore data library, on the ICRU73 data tables or on the Penelope2008 Monte Carlo code. They adopt the same software design as the "standard" GEANT4 electromagnetic models.

Examples of the registration of physics constructor with low-energy electromagnetic models are shown in GEANT4 extended examples (`examples/extended/electromagnetic` and `examples/extended/medical` in the GEANT4 source distribution). Advanced examples (`examples/advanced` in the GEANT4 source distribution) illustrate alternative instantiation of these processes. Both are available as part of the GEANT4 release.

### Production Cuts

Remember that production cuts for secondaries can be specified as range cuts, which are converted at initialisation time into energy thresholds for secondary gamma, electron, positron and proton production. The cut for proton is applied by elastic scattering processes to all recoil ions.

A range cut value is set by default to 0.7 mm in GEANT4 reference physics lists. This value can be specified in the optional SetCuts() method of the user Physics list or via UI commands. For e.g. to set a range cut of 10 micrometers, one can use

```
/run/setCut  0.01 mm
```

or, for a given particle type (for e.g. electron),

```
/run/setCutForAGivenParticle e- 0.01 mm
```

If a range cut equivalent to an energy lower than 990 eV is specified, the energy cut is still set to 990 eV. In order to decrease this value (for e.g. down to 250 eV, in order to simulate low energy emission lines of the fluorescence spectrum), one may use the following UI command before the "/run/initialize" command

```
/cuts/setLowEdge 100 eV
```

or alternatively directly in the user Physics list, in the optional SetCuts() method, using:

```
G4ProductionCutsTable::GetProductionCutsTable()->SetEnergyRange(100*eV, 1*GeV);
```

A command is also available in order to disable usage of production threshold for fluorescence and Auger electron production

```
/process/em/deexcitationIgnoreCut true
```

### Angular Generators

For part of EM processes it is possible to factorise out sampling of secondary energy and direction. Using an interface `G4VEmModel` base class `SetAngularDistribution(G4VEmAngularDistribution*)` it is possible to substitute default angular generator of a model. Angular generators in standard and lowenergy sub-packages follow the same abstract interface.

For photoelectric models several angular generators are available:

- G4SauterGavrilaAngularDistribution (default);
- G4PhotoElectricAngularGeneratorSauterGavrila;
- G4PhotoElectricAngularGeneratorPolarized.

For bremsstrahlung and pair production models following angular generators are available:

- G4ModifiedTsai (default for electrons and positrons);
- G4ModifiedMephi (default for muons and hadrons);
- G4DipBustGenerator;
- G4Generator2BS (recommended for electrons and positrons);
- G4Generator2BN;
- G4PenelopeBremsstrahlungAngular.

For gamma conversion models following angular generators are available:

- G4ModifiedTsai (default);
- G4DipBustGenerator.

For models of ionisation a new optional angular generator is available:

- G4DeltaAngle.

### Electromagnetics secondary biasing

It may be useful to create more than one secondary at an interaction. For example, electrons incident on a target in a medical linac produce photons through bremsstrahlung. The variance reduction technique of bremsstrahlung splitting involves choosing *N* photons from the expected distribution, and assigning each a weight of 1/*N*.

Similarly, if the secondaries are not important, one can kill them with a survival probability of 1/*N*. The weight of the survivors is increased by a factor *N*. This is known as Russian roulette.

Neither biasing technique is applied if the resulting daughter particles would have a weight below 1/*N*, in the case of brem splitting, or above 1, in the case of Russian roulette.

These techniques can be enabled in GEANT4 electromagnetics with the macro commands

```
/process/em/setSecBiasing processName Region factor energyLimit energyUnit
```

where: *processName* is the name of the process to apply the biasing to; *Region* is the region in which to apply biasing; *factor* is the inverse of the brem splitting or Russian roulette factor (1/*N*); *energyLimit energyUnit* is the high energy limit. If the first secondary has energy above this limit, no biasing is applied.

For example,

```
/process/em/setSecBiasing eBrem target 10 100 MeV
```

will result in electrons undergoing bremsstrahlung in the target region being split 10 times (if the first photon sampled has an energy less than 100 MeV).

Note that the biasing needs to be specified for each process individually. To apply Russian Roulette to daughter electrons from interactions of photons, issue the macro command for the processes phot, compt, conv.

### Directional splitting

This biasing may be enabled based on the direction of the outgoing particles ("directional splitting"). The user may specify a spherical volume of interest by giving the center and radius of the volume. In an interaction, if the incident particle has high weight, the outgoing particles are split. *N* particles from the distribution are created, each with weight 1/*N*. For each particle, if it is not directed towards the volume of interest, Russian Roulette is played. Typically one will want directional splitting to take place for all interactions.

For example,

```
/process/em/setDirectionalSplitting true
/process/em/setDirectionalSplittingTarget 1000 0 0 mm  # x, y, z components of center
/process/em/setDirectionalSplittingRadius 10 cm
/process/em/setSecBiasing eBrem world 100 100 MeV
/process/em/setSecBiasing Rayl world 100 100 MeV
/process/em/setSecBiasing phot world 100 100 MeV
/process/em/setSecBiasing compt world 100 100 MeV
/process/em/setSecBiasing annihil world 100 100 MeV
```

Reference: BEAMnrc Users Manual, D.W.O Rogers, B. Walters, I. Kawrakow. NRCC Report PIRS-0509(A)revL, available here.

### Livermore Data Based Models

- **Photon models**
  - Photo-electric effect (class `G4LivermorePhotoElectricModel`)
  - Polarized Photo-electric effect (class `G4LivermorePolarizedPhotoElectricModel`)
  - Compton scattering (class `G4LivermoreComptonModel`)
  - Compton scattering (class `G4LowEPComptonModel`)
  - Polarized Compton scattering (class `G4LivermorePolarizedComptonModel`)
  - Rayleigh scattering (class `G4LivermoreRayleighModel`)
  - Polarized Rayleigh scattering (class `G4LivermorePolarizedRayleighModel`)
  - Gamma conversion (also called pair production, class `G4LivermoreGammaConversionModel`)
  - Nuclear gamma conversion (class `G4LivermoreNuclearGammaConversionModel`)
  - Polarized gamma conversion (class `G4LivermorePolarizedGammaConversionModel`)
- **Electron models**
  - Bremsstrahlung (class `G4LivermoreBremsstrahlungModel`)
  - Ionisation and delta ray production (class `G4LivermoreIonisationModel`)

### Hadron and Ion Ionisation Models

Ionisation and delta ray production by hadrons and ions base on stopping power data at low energies (below 2 MeV/u) and Bethe-Bloch or Lindhard-Sorensen theories above (*J. Lindhard & A.H. Sorensen, Phys. Rev. A 53 (1996) 2443-2455*). The data are taken from ICRU90, ICRU73, PSTAR, ASTAR, and ICRU49 databases. Part of these data are transformed to G4LEDATA database of Geant4, the rest is hardcoded inside corresponding GEANT4 classes. ICRU90 provides new accurate data but only for 3 target materials and limited number of projectile/target combinations. ICRU73 cover projectile/target ion couple fro Z=3 to Z=80. ASTAR is used only for Helium ions.

### Penelope2008 Based Models

- **Photon models**
  - Compton scattering (class `G4PenelopeComptonModel`)
  - Rayleigh scattering (class `G4PenelopeRayleighModel`)
  - Gamma conversion (also called pair production, class `G4PenelopeGammaConversionModel`)
  - Photo-electric effect (class `G4PenelopePhotoElectricModel`)
- **Electron models**
  - Bremsstrahlung (class `G4PenelopeBremsstrahlungModel`)
  - Ionisation and delta ray production (class `G4PenelopeIonisationModel`)
- **Positron models**
  - Bremsstrahlung (class `G4PenelopeBremsstrahlungModel`)
  - Ionisation and delta ray production (class `G4PenelopeIonisationModel`)
  - Positron annihilation (class `G4PenelopeAnnihilationModel`)

All Penelope models can be applied up to a maximum energy of 100 GeV, although it is advisable not to use them above a few hundreds of MeV.

Options are available in the all Penelope Models, allowing to set (and retrieve) the verbosity level of the model, namely the amount of information which is printed on the screen.

- SetVerbosityLevel(G4int)
- GetVerbosityLevel()

The default verbosity level is 0 (namely, no textual output on the screen). The default value should be used in general for normal runs. Higher verbosity levels are suggested only for testing and debugging purposes.

The verbosity scale defined for all Penelope processes is the following:

- 0 = no printout on the screen (default)

- 1 = issue warnings only in the case of energy non-conservation in the final state (should never happen)
- 2 = reports full details on the energy budget in the final state
- 3 = writes also information on cross section calculation, data file opening and sampling of atoms
- 4 = issues messages when entering in methods

## Very Low energy Electromagnetic Processes (GEANT4-DNA extension)

The GEANT4 low energy electromagnetic Physics package has been extended down to energies of a few electron Volts suitable for the simulation of radiation effects in liquid water for applications in micro/nanodosimetry at the cellular and sub-cellular level. These developments take place in the framework of the on-going GEANT4-DNA project (see more in the Geant4-DNA web pages or in the EM web pages of the Geant4 Electromagnetic Physics Working Group).

The GEANT4 -DNA process and model classes apply to electrons, protons, hydrogen, alpha particles and their charge states, in liquid water (“G4_WATER” material).

**Electron processes and models**

- Elastic scattering:
  - process class is G4DNAElastic
  - four alternative model classes are: G4DNAScreenedRutherfordElasticModel or G4DNAChampionElasticModel (default) or G4DNAUeharaScreenedRutherfordElasticModel or G4DNACPA100ElasticModel
- Excitation
  - process class is G4DNAExcitation
  - model class is G4DNABornExcitationModel (default) or G4DNAEmfietzoglouExcitationModel or G4DNACPA100ExcitationModel
- Ionisation
  - process class is G4DNAIonisation
  - model class is G4DNABornIonisationModel (default) or G4DNAEmfietzoglouIonisationModel or G4DNACPA100IonisationModel
- Attachment
  - process class is G4DNAAttachment
  - model class is G4DNAMeltonAttachmentModel
- Vibrational excitation
  - process class is G4DNAVibExcitation
  - model class is G4DNASancheExcitationModel

**Proton processes and models**

- Elastic scattering:
  - process class is G4DNAElastic
  - G4DNAIonElasticModel
- Excitation
  - process class is G4DNAExcitation
  - two complementary model classes are G4DNAMillerGreenExcitationModel (below 500 keV) and G4DNABornExcitationModel (above)
- Ionisation
  - process class is G4DNAIonisation
  - two complementary model classes are G4DNARuddIonisationExtendedModel (below 500 keV) and G4DNABornIonisationModel (above)
- Charge decrease
  - process class is G4DNAChargeDecrease
  - model class is G4DNADingfelderChargeDecreaseModel

**Hydrogen processes and models**

- Elastic scattering :

  - process class is G4DNAElastic
  - G4DNAIonElasticModel
- Excitation
  - process class is G4DNAExcitation
  - model class is G4DNAMillerGreenExcitationModel
- Ionisation
  - process class is G4DNAIonisation
  - model class is G4DNARuddIonisationModel
- Charge increase
  - process class is G4DNAChargeIncrease
  - model class is G4DNADingfelderChargeIncreaseModel

**Helium (neutral) processes and models**

- Elastic scattering :
  - process class is G4DNAElastic
  - G4DNAIonElasticModel
- Excitation
  - process class is G4DNAExcitation
  - model class is G4DNAMillerGreenExcitationModel
- Ionisation
  - process class is G4DNAIonisation
  - model class is G4DNARuddIonisationModel
- Charge increase
  - process class is G4DNAChargeIncrease
  - model class is G4DNADingfelderChargeIncreaseModel

**Helium+ (ionized once) processes and models**

- Elastic scattering :
  - process class is G4DNAElastic
  - G4DNAIonElasticModel
- Excitation
  - process class is G4DNAExcitation
  - model class is G4DNAMillerGreenExcitationModel
- Ionisation
  - process class is G4DNAIonisation
  - model classes is G4DNARuddIonisationModel
- Charge increase
  - process class is G4DNAChargeIncrease
  - model classes is G4DNADingfelderChargeIncreaseModel
- Charge decrease
  - process class is G4DNAChargeDecrease
  - model classes is G4DNADingfelderChargeDecreaseModel

**Helium++ (ionised twice) processes and models**

- Elastic scattering :
  - process class is G4DNAElastic
  - G4DNAIonElasticModel
- Excitation
  - process class is G4DNAExcitation
  - model classes is G4DNAMillerGreenExcitationModel
- Ionisation
  - process class is G4DNAIonisation
  - model classes is G4DNARuddIonisationModel
- Charge decrease

  - process class is G4DNAChargeDecrease
  - model classes is G4DNADingfelderChargeDecreaseModel

** Ion processes and models**

- Ionisation
  - process class is G4DNAIonisation
  - model class is G4DNARuddIonisationExtendedModel

Examples of the registration of these processes in a physics list are given in the G4EmDNAPhysics* constructors (in `source/physics_lists/constructors/electromagnetic` in the GEANT4 source distribution). An example of the usage of these constructors in a physics list is given in the "dnaphysics" extended example, which explains how to extract basic information from GEANT4-DNA Physics processes.

GEANT4-DNA physics constructors are described at the Geant4-DNA website.

The "microdosimetry" extended example illustrates how to combine GEANT4-DNA processes with Standard electromagnetic processes (combination of discrete and condensed history GEANT4 electromagnetic processes at different scales).

A set of GEANT4-DNA models applicable to biological materials is available since release 10.4. These models are named G4DNAPTBElasticModel, G4DNAPTBExcitationModel, G4DNAPTBIonisationModel and G4DNAPTBAugerModel. They can be used for electrons in THF, PY, PU, TMP precursors and in backbone, cytosine, thymine, adenine, guanine materials of DNA. The G4DNAPTBIonisationModel can also be used with protons in THF, PY and TMP. Their usage is illustrated in the "icsd" extended example.

Since GEANT4 release 11.0, GEANT4-DNA can be used to describe electron interactions down to 10 eV in gold using a tracking structure approach. Electron processes and models include:

- Elastic scattering
  - process class is G4DNAElastic
  - model class is G4DNAELSEPAElasticModel
- Ionization
  - process class is G4DNAIonisation
  - model class is G4DNARelativisticIonisationModel
- Excitation
  - process class is G4DNAExcitation
  - model class is G4DNADiracRMatrixExcitationModel
- Plasmon excitation
  - process class is G4DNAPlasmonExcitation
  - model class is G4DNAQuinnPlasmonExcitationModel

See more details on these physics models in:

*D. Sakata et al., An implementation of discrete electron transport models for gold in the Geant4 simulation toolkit, Journal of Applied Physics, 120, 244901, 2016*, linked here

Since GEANT4 release 10.1, GEANT4-DNA can also be used for the modelling of water radiolysis (physico-chemistry and chemistry stages). Three extended examples, "chem1", "chem2", "chem3" and "chem4" illustrate this. More information is available from the Geant4-DNA website.

To run the GEANT4-DNA extension, data files need to be copied by the user to his/her code repository. These files are distributed together with the GEANT4 release. The user should set the environment variable G4LEDATA to the directory where he/she has copied the files.

A full list of publications regarding GEANT4-DNA is directly available from the Geant4-DNA website or from the Geant4@IN2P3 web site).

### Atomic Deexcitation

A unique interface named G4VAtomicDeexcitation is available in GEANT4 for the simulation of atomic deexcitation using Standard, Low Energy and Very Low Energy electromagnetic processes. Atomic deexcitation includes fluorescence and Auger electron emission induced by photons, electrons and ions (PIXE); see more details in:

A. Mantero et al., PIXE Simulation in Geant4 , X-Ray Spec. , 40, 135-140, 2011.

It can be activated for processes producing vacancies in atomic shells. Currently these processes are the photoelectric effect, ionization and Compton scattering.

**Activation of atomic deexcitation**

The activation of atomic deexcitation in continuous processes in a user physics list can be done through the following G4EmParameters class methods described above or via UI commands

```
/process/em/deexcitation region true true true
/process/em/fluo true
/process/em/auger true
/process/em/pixe true
```

One can define parameters in the G4State_PreInit or G4State_Idle states. Fluorescence from photons and electrons is activated by default in Option3, Option4, Livermore and Penelope physics constructors, while Auger production and PIXE are not.

The alternative set of data by Bearden et al. (1967) for the modelling of fluorescence lines had been added to the G4LEDATA archive. This set can be selected via UI command

```
/process/em/fluoDirectory name
```

Another important UI commands enable simulation of the full Auger and/or fluorescence cascade

```
/process/em/deexcitationIgnoreCut true
```

**How to change ionisation cross section models ?**

The user can also select which cross section model to use in order to calculate shell ionisation cross sections for generating PIXE

```
/process/em/pixeXSmodel     name
/process/em/pixeElecXSmodel name
```

where the name can be “Empirical”, “ECPSSR_FormFactor”, “ECPSSR_Analytical” or “ECPSSR_ANSTO” corresponds to different PIXE cross sections. Following shell cross sections models are available : “ECPSSR_Analytical” models derive from an analytical calculation of the ECPSSR theory (see *A. Mantero et al., X-Ray Spec.40 (2011) 135-140*) and it reproduces K and L shell cross sections over a wide range of energies; “ECPSSR_FormFactor” models derive from A. Taborda et al. calculations (see *A. Taborda et al., X-Ray Spec. 40 (2011) 127-134*) of ECPSSR values directly form Form Factors and it covers K, L shells on the range 0.1-100 MeV and M shells in the range 0.1-10 MeV; the “empirical” models are from Paul “reference values” (for protons and alphas for K-Shell) and Orlic empirical model for L shells (only for protons and ions with Z>2). The later ones are the models used by default. Out of the energy boundaries, “ECPSSR_Analytical” model is used. We recommend to use default settings if not sure what to use. “ECPSSR_ANSTO” models are ECPSSR calculations based on state of the art recommendations documented in *D. Cohen et al., K, L, and M shell datasets for PIXE spectrum fitting and analysis, NIM B, 363, 7-18, 2015*, linked here

**Example**

The **TestEm5 extended/electromagetic example** shows how to simulate atomic deexcitation (see for e.g. the pixe.mac and pixe_ANSTO macros).

### Very Low energy Electromagnetic Processes in Silicon for microelectronics application (GEANT4-MuElec extension)

(Previously named GEANT4-MuElec)

The GEANT4 low energy electromagnetic Physics package has been extended down to energies of a few electron Volts suitable for the simulation of radiation effects in highly integrated microelectronic components.

The GEANT4-MicroElec process and model classes apply to electrons, protons and heavy ions in silicon.

**Electron processes and models**

- Elastic scattering :
  - process class is G4MicroElastic
  - model class is G4MicroElecElasticModel_new
- Ionization
  - process class is G4MicroElecInelastic
  - model class is G4MicroElecInelasticModel_new

**Proton processes and models**

- Ionisation
  - process class is G4MicroElecInelastic
  - model class is G4MicroElecInelasticModel_new

**Heavy ion processes and models**

- Ionization
  - process class is G4MicroElecInelastic
  - model class is G4MicroElecInelasticModel_new

A full list of publications regarding GEANT4-MicroElec is directly available from the Geant4-MicroElec website.

### New Compton model by Monash U., Australia

A new Compton scattering model for unpolarised photons has been developed in the relativistic impulse approximation. The model was developed as an alternative to low energy electromagnetic Compton scattering models developed from Ribberfors' Compton scattering framework (Livermore, Penelope Compton models). The model class is named named G4LowEPComptonModel.

G4LowEPComptonModel has been added to the physics constructor G4EmStandardPhysics_option4, containing the most accurate models from the Standard and Low Energy Electromagnetic physics working groups.

### Multi-scale Processes

### Hadron Impact Ionisation and PIXE

The **G4hImpactIonisation** process deals with ionisation by impact of hadrons and alpha particles, and the following generation of **PIXE** (Particle Induced X-ray Emission). This process and related classes can be found in *source/processes/electromagnetic/pii*.

Further documentation about PIXE simulation with this process is available here.

A detailed description of the related physics features can be found in:

M. G. Pia et al., PIXE Simulation with Geant4 , IEEE Trans. Nucl. Sci. , vol. 56, no. 6, pp. 3614-3649, 2009.

A brief summary of the related physics features can be found in the GEANT4 Physics Reference Manual.

An example of how to use this process is shown below. A more extensive example is available in the eRosita GEANT4 advanced example (see *examples/advanced/eRosita* in your GEANT4 installation source).

```
#include "G4hImpactIonisation.hh"
[...]

void eRositaPhysicsList::ConstructProcess()
{

[...]

  theParticleIterator->reset();
  while( (*theParticleIterator)() )
    {
      G4ParticleDefinition* particle = theParticleIterator->value();
      G4ProcessManager* processManager = particle->GetProcessManager();
      G4String particleName = particle->GetParticleName();

      if (particleName == "proton")
        {
          // Instantiate the G4hImpactIonisation process
          G4hImpactIonisation* hIonisation = new G4hImpactIonisation();

          // Select the cross section models to be applied for K, L and M shell vacancy creation
          // (here the ECPSSR model is selected for K, L and M shell; one can mix and match
          // different models for each shell)
          hIonisation->SetPixeCrossSectionK("ecpssr");
          hIonisation->SetPixeCrossSectionL("ecpssr");
          hIonisation->SetPixeCrossSectionM("ecpssr");

          // Register the process with the processManager associated with protons
          processManager -> AddProcess(hIonisation, -1, 2, 2);
        }
    }
}
```

**Available cross section model options**

The following cross section model options are available:

- protons
  - K shell
    - `ecpssr` *(based on the ECPSSR theory)*
    - `ecpssr_hs` *(based on the ECPSSR theory, with Hartree-Slater correction)*
    - `ecpssr_ua` *(based on the ECPSSR theory, with United Atom Hartree-Slater correction)*
    - `ecpssr_he` *(based on the ECPSSR theory, with high energy correction)*
    - `pwba` *(plane wave Born approximation)*
    - `paul` *(based on the empirical model by Paul and Sacher)*
    - `kahoul` *(based on the empirical model by Kahoul et al.)*
  - L shell
    - `ecpssr`
    - `ecpssr_ua`
    - `pwba`
    - `miyagawa` *(based on the empirical model by Miyagawa et al.)*
    - `orlic` *(based on the empirical model by Orlic et al.)*
    - `sow` *(based on the empirical model by Sow et al.)*
  - M shell
    - `ecpssr`
    - `pwba`
- alpha particles
  - K shell
    - `ecpssr`

    * `ecpssr_hs`
    * `pwba`
    * `paul` *(based on the empirical model by Paul and Bolik)*
  - L shell
    * `ecpssr`
    * `pwba`
  - M shell
    * `ecpssr`
    * `pwba`

**PIXE data library**

The *G4hImpactIonisation* process uses a **PIXE Data Library.**

The PIXE Data Library is distributed in the GEANT4 **G4PII** data set, which must be downloaded along with GEANT4 source code.

The **G4PIIDATA** environment variable must be defined to refer to the location of the G4PII PIXE data library in your filesystem; for instance, if you use a c-like shell

```
setenv G4PIIDATA path_to_where_G4PII_has_been_downloaded
```

Further documentation about the PIXE Data Library is available here.

## 5.2.3 Hadronic Interactions

This section briefly introduces the hadronic physics processes installed in GEANT4. For details of the implementation of hadronic interactions available in GEANT4, please refer to the Physics Reference Manual.

### Treatment of Cross Sections

#### Cross section data sets

Each hadronic process object (derived from `G4HadronicProcess`) may have one or more cross section data sets associated with it. The term "data set" is meant, in a broad sense, to be an object that encapsulates methods and data for calculating total cross sections for a given process. The methods and data may take many forms, from a simple equation using a few hard-wired numbers to a sophisticated parameterisation using large data tables. Cross section data sets are derived from the abstract class `G4VCrossSectionDataSet`, and are required to implement the following methods:

```
G4bool IsApplicable( const G4DynamicParticle*, const G4Element* )
```

This method must return `True` if the data set is able to calculate a total cross section for the given particle and material, and `False` otherwise.

```
G4double GetCrossSection( const G4DynamicParticle*, const G4Element* )
```

This method, which will be invoked only if `True` was returned by `IsApplicable`, must return a cross section, in GEANT4 default units, for the given particle and material.

```
void BuildPhysicsTable( const G4ParticleDefinition& )
```

This method may be invoked to request the data set to recalculate its internal database or otherwise reset its state after a change in the cuts or other parameters of the given particle type.

```
void DumpPhysicsTable( const G4ParticleDefinition& )
```

This method may be invoked to request the data set to print its internal database and/or other state information, for the given particle type, to the standard output stream.

### Cross section data store

Cross section data sets are used by the process for the calculation of the physical interaction length. A given cross section data set may only apply to a certain energy range, or may only be able to calculate cross sections for a particular type of particle. The class `G4CrossSectionDataStore` has been provided to allow the user to specify, if desired, a series of data sets for a process, and to arrange the priority of data sets so that the appropriate one is used for a given energy range, particle, and material. It implements the following public method:

```
G4double ComputeCrossSection( const G4DynamicParticle*, const G4Material* )
G4double GetCrossSection( const G4DynamicParticle*, const G4Element*, const G4Material* )
```

For a given particle and material, this method returns a cross section value provided by one of the collection of cross section data sets listed in the data store object. If there are no known data sets, a `G4Exception` is thrown and `DBL_MIN` is returned. Otherwise, each data set in the list is queried, in reverse list order, by invoking its `IsApplicable` method for the given particle and material. The first data set object that responds positively will then be asked to return a cross section value via its `GetCrossSection` method. If no data set responds positively, a `G4Exception` is thrown and `DBL_MIN` is returned.

```
void AddDataSet( G4VCrossSectionDataSet* aDataSet )
```

This method adds the given cross section data set to the end of the list of data sets in the data store. For the evaluation of cross sections, the list has a LIFO (Last In First Out) priority, meaning that data sets added later to the list will have priority over those added earlier to the list. Another way of saying this, is that the data store, when given a `GetCrossSection` request, does the `IsApplicable` queries in the reverse list order, starting with the last data set in the list and proceeding to the first, and the first data set that responds positively is used to calculate the cross section.

```
void BuildPhysicsTable( const G4ParticleDefinition& aParticleType )
```

This method may be invoked to indicate to the data store that there has been a change in the cuts or other parameters of the given particle type. In response, the data store will invoke the `BuildPhysicsTable` of each of its data sets.

```
void DumpPhysicsTable( const G4ParticleDefinition& )
```

This method may be used to request the data store to invoke the `DumpPhysicsTable` method of each of its data sets.

### Default cross sections

The default cross section for main hadrons:

```
-  G4ParticleInelasticXS and G4BGGNucleonElasticXS for protons
-  G4NeutronInelasticXS, G4NeutronElasticXS, G4NeutronCaptureXS
-  G4BGGPionInelasticXS and G4BGGPionElasticXS for pions
-  G4GammaNuclearXS for gamma-nuclear
```

For other hadrons and ions cross section is provided by generic classes:

```
-  G4CrosssectionInelastic
-  G4CrossSectionElastic
```

which require concrete cross section implementation via the interface G4VComponentCrossSection. Following main components are used:

```
-  G4ComponentGGHadronNucleusXsc for hadrons
-  G4ComponentAntiNuclNuclearXS for anti-protons and anti light ions
-  G4ComponentGGNuclNuclXsc for ions
```

The default cross sections can be overridden in whole or in part by the user. To this end, the base class G4HadronicProcess has a get method:

```
G4CrossSectionDataStore* GetCrossSectionDataStore()
```

which gives public access to the data store for each process. The user's cross section data sets can be added to the data store according to the following framework:

```
G4Hadron...Process aProcess(...)

MyCrossSectionDataSet myDataSet(...)

aProcess.GetCrossSectionDataStore()->AddDataSet( &MyDataSet )
```

The added data set will override the default cross section data whenever so indicated by its IsApplicable method.

In addition to the get method, G4HadronicProcess also has the method

```
void SetCrossSectionDataStore( G4CrossSectionDataStore* )
```

which allows the user to completely replace the default data store with a new data store.

It should be noted that a process does not send any information about itself to its associated data store (and hence data set) objects. Thus, each data set is assumed to be formulated to calculate cross sections for one and only one type of process. Of course, this does not prevent different data sets from sharing common data and/or calculation methods, as in the case of the G4HadronCrossSections class mentioned above. Indeed, G4VCrossSectionDataSet specifies only the abstract interface between physics processes and their data sets, and leaves the user free to implement whatever sort of underlying structure is appropriate.

### Cross-sections for low energy neutron transport

The cross section data for low energy neutron transport are organized in a set of files that are read in by the corresponding data set classes at time zero. Hereby the file system is used, in order to allow highly granular access to the data. The ``root" directory of the cross-section directory structure is accessed through an environment variable, G4NEUTRONHPDATA, which is to be set by the user. The classes accessing the total cross-sections of the individual processes, i.e., the cross-section data set classes for low energy neutron transport, are G4NeutronHPElasticData, G4NeutronHPCaptureData, G4NeutronHPFissionData, and G4NeutronHPInelasticData.

For detailed descriptions of the low energy neutron total cross-sections, they may be registered by the user as described above with the data stores of the corresponding processes for neutron interactions.

It should be noted that using these total cross section classes does not require that the neutron_hp models also be used. It is up to the user to decide whether this is desirable or not for his particular problem.

The compact version of neutron cross sections derived from HP database are provided with classes G4NeutronInelasticXS, G4NeutronElasticXS, and G4NeutronCaptureXS. Using low-energy data for protons, deuterons, tritons, alpha, and He3 the data for the class G4ParticleInelasticXS are obtained. Gamma-nuclear cross section is extracted from the IAEA Evaluated Photonuclear Data Library (IAEA/PD-2019) here.

These cross-sections for n, p, d, t, He3, He4, and gamma are accessed through an environment variable G4PARTICLEXSDATA.

### Cross-sections for low-energy charged particle transport

The cross-section data for low-energy charged particle transport are organized in a set of files that are read at initialization, similarly to the case of low-energy neutron transport. The "root" directory of the cross-section directory structure is accessed through an environment variable, `G4PARTICLEHPDATA`, which has to be set by the user. This variable has to point to the directory where the low-energy charged particle data have been installed, e.g. `G4TENDL1.4` for the GEANT4 release `10.7` (note that the download of this data library from the GEANT4 web site is not done automatically, i.e. it must be done manually by the user):

```
export G4PARTICLEHPDATA=/your/path/G4TENDL1.4/
```

It is expected that the directory `$G4PARTICLEHPDATA` has the following five subdirectories, corresponding to the charged particles that can be handled by the low-energy charged particle transport: `Proton/`, `Deuteron/`, `Triton/`, `He3/`, `Alpha/`. It is possible for the user to overwrite the default directory structure with individual environment variables pointing to custom data libraries for each particle type. This is considered an advanced/expert user feature. These directories are set by the following environment variables: `G4PROTONHPDATA`, for proton; `G4DEUTERONHPDATA`, for deuteron; `G4TRITONHPDATA`, for triton; `G4HE3HPDATA`, for He3; `G4ALPHAHPDATA`, for alpha. Note that if any of these variables is not defined explicitly, e.g. `G4TRITONHPDATA`, then the corresponding data library is expected to be a subdirectory of `$G4PARTICLEHPDATA/`, e.g. `$G4PARTICLEHPDATA/Triton/`. If instead all the above five environmental variables are set, then `G4PARTICLEHPDATA` does not need to be specified; even if it is set, then its value will be ignored (because the per-particle ones take precedence).

## Hadrons at Rest

### List of implemented "Hadron at Rest" processes

The following process classes have been implemented:

- $\pi^-, K^-, \sigma^-, \xi^-, \omega^-$ absorption (class name `G4HadronicAbsorptionBertini`)
- neutron capture (class name `G4NeutronCaptureProcess`)
- anti-proton, anti-$\sigma^+$, anti-deuteron, anti-triton, anti-alpha, anti-He3 annihilation (class name `G4HadronicAbsorptionFritiof`)
- mu- capture (class name `G4MuonMinusCapture`)

Capture of low-energy negatively charged particles is a complex process involving formation of mesonic atoms, X-ray cascade and Auger cascade, nuclear interaction. In the case of mu- there is also a probability to decay from K-shell of mesonic atom. To handle this a base process class `G4HadronicStoppingProcess` is used.

For the case of neutrons, GEANT4 offer simulation down to thermal energies. The capture cross section generally increases when neutron energy decreases and there are many nuclear resonances. In GEANT4 neutron capture cross sections are parameterized using ENDF database.

## Hadrons in Flight

### What processes do you need?

For hadrons in motion, there are four physics process classes. Table 5.1 shows each process and the particles for which it is relevant.

Table 5.1: Hadronic processes and relevant particles.

| | |
|---|---|
| `G4HadronElasticProcess` | $\pi^+, \pi^-, K^+, K_S^0, K_L^0, K^-, p, \bar{p}, n, \bar{n}, \lambda, \bar{\lambda}, \Sigma^+, \Sigma^-, \bar{\Sigma}^+, \bar{\Sigma}^-, \Xi^0, \Xi^-, \bar{\Xi}^0, \bar{\Xi}^-$ |
| `G4HadronInelasticProcess` | $\pi^+, \pi^-, K^+, K_S^0, K_L^0, K^-, p, \bar{p}, n, \bar{n}, \lambda, \bar{\lambda}, \Sigma^+, \Sigma^-, \bar{\Sigma}^+, \bar{\Sigma}^-, \Xi^0, \Xi^-, \bar{\Xi}^0, \bar{\Xi}^-$ |
| `G4NeutronFissionProcess` | neutron |
| `G4NeutronCaptureProcess` | neutron |

## How to register Models

To register an inelastic process model for a particle, a proton for example, first get the pointer to the particle's process manager:

```
G4ParticleDefinition *theProton = G4Proton::ProtonDefinition();
G4ProcessManager *theProtonProcMan = theProton->GetProcessManager();
```

Create an instance of the particle's inelastic process:

```
G4HadronicProcess *theProcess = new G4HadronicProcess();
```

Create an instance of the model which determines the secondaries produced in the interaction, and calculates the momenta of the particles, for instance the Bertini cascade model:

```
G4CascadeInterface *theCascade = new G4CascadeInterface();
```

Register the model with the particle's inelastic process:

```
theProcess->RegisterMe( theCascade );
```

Finally, add the particle's inelastic process to the list of discrete processes:

```
theProcessManager->AddDiscreteProcess( theProcess );
```

The particle's inelastic process class, `G4HadronInelasticProcess` may be used, which is equivalent to `G4HadronicProcess` class. The `G4HadronicProcess` class derives from the `G4VDiscreteProcess` class. The inelastic, elastic, capture, and fission processes derive from the `G4HadronicProcess` class. This pure virtual class also provides the energy range manager object and the `RegisterMe` access function.

In-flight, final-state hadronic models derive, directly or indirectly, from the `G4InelasticInteraction` class, which is an abstract base class since the pure virtual function `ApplyYourself` is not defined there. `G4InelasticInteraction` itself derives from the `G4HadronicInteraction` abstract base class. This class is the base class for all the model classes. It sorts out the energy range for the models and provides class utilities. The `G4HadronicInteraction` class provides the `Set/GetMinEnergy` and the `Set/GetMaxEnergy` functions which determine the minimum and maximum energy range for the model. An energy range can be set for a specific element, a specific material, or for general applicability:

```
void SetMinEnergy( G4double anEnergy, G4Element *anElement )
void SetMinEnergy( G4double anEnergy, G4Material *aMaterial )
void SetMinEnergy( const G4double anEnergy )
void SetMaxEnergy( G4double anEnergy, G4Element *anElement )
void SetMaxEnergy( G4double anEnergy, G4Material *aMaterial )
void SetMaxEnergy( const G4double anEnergy )
```

### Which models are there, and what are the defaults

In GEANT4, any model can be run together with any other model without the need for the implementation of a special interface, or batch suite, and the ranges of applicability for the different models can be steered at initialisation time. This way, highly specialised models (valid only for one material and particle, and applicable only in a very restricted energy range) can be used in the same application, together with more general code, in a coherent fashion.

Each model has an intrinsic range of applicability, and the model chosen for a simulation depends very much on the use-case. Consequently, there are no "defaults". However, physics lists are provided which specify sets of models for various purposes.

Two types of hadronic shower models have been implemented: data driven models and theory driven models.

- Data driven models are available for the transport of low energy neutrons in matter in sub-directory `hadronics/models/neutron_hp`. The modeling is based on the data formats of **ENDF/6**, and all distributions of this standard data format are implemented. The data sets used are selected from data libraries that conform to these standard formats. The file system is used in order to allow granular access to, and flexibility in, the use of the cross sections for different isotopes, and channels. The energy coverage of these models is from thermal energies to 20 MeV.
- Theory driven models are available for inelastic scattering in a first implementation, covering the full energy range of LHC experiments. They are located in sub-directory `hadronics/models/generator`. The current philosophy implies the usage of parton string models at high energies, of intra-nuclear transport models at intermediate energies, and of statistical break-up models for de-excitation.

### High-precision neutron interactions (NeutronHP)

Nuclear models fail (sometimes catastrophically) at predicting with reasonable accuracies the nuclear cross sections of neutrons (and other particles). For this reason, all physical quantities relevant for an accurate modeling of nuclear reactions in Monte Carlo simulations need to be provided as a database which includes, ideally:

- cross sections
- angular distributions of the emitted particles
- energy spectra of the emitted particles
- energy-angle correlated spectrum (double-differential cross sections, DDX)
- neutrons per fission
- fission spectra
- fission product yields
- photo production data

For the case of neutron induced reactions, such databases are called "evaluated data", in the sense that they contain recommended values for different quantities that rely on compilations of experimental nuclear data and usually completed with theoretical predictions, benchmarked against available experimental data (i.e. integral and differential experiments) when possible. It should be noticed that the information available varies from isotope to isotope and can be incomplete or totally missing.

The G4NeutronHP package in GEANT4 allows using evaluated nuclear data libraries in the G4NDL format. GEANT4 users should know that any simulation involving neutrons with energies below 20 MeV and not using the G4NeutronHP package can lead to unreliable results. GEANT4 users are therefore encouraged to use it, although they should be aware of the limitations of using evaluated nuclear data libraries.

An example about how to implement the G4NeutronHP package into physics list in a GEANT4 application can be found in the example case (among others distributed with GEANT4) `extended/radioactivedecay/rdecay02`. Three different processes are included in that example: elastic, capture and inelastic. The inelastic reactions in G4NeutronHP are all reactions except elastic, capture and fission, so fission should also be included in the physics list, if needed, and it is done in the same way as it is done for the other three.

The G4NeutronHP package must be used together with evaluated nuclear data libraries. They are available on the GEANT4 download page and from the IAEA nuclear data web site where a larger set of different libraries, including isotopes with Z > 92, is available.

The evaluated nuclear data libraries do differ and thus the results of the Monte Carlo simulations will depend on the library used. It is a safe practice to perform simulations with (at least) two different libraries for estimating the uncertainties associated to the nuclear data.

Together with a good implementation of the physics list, users must be very careful with the definition of the materials performed in a Monte Carlo simulation when low energy neutron transport is relevant. In contrast to other kind of simulations, the isotopic composition of the elements which compose the different materials can strongly affect the obtained simulation results. Because of this, it is strongly recommended to define specifically the isotopic composition of each element used in the simulation, as it is described in the GEANT4 user's manual. In principle, such a practice is not mandatory if natural isotopic compositions are used, since GEANT4 contains them in their databases. However, by defining them explicitly some unexpected problems may be avoided and a better control of the simulation will be achieved.

It is highly recommended or mandatory to set the following UNIX environment variable and UI commands when running a GEANT4 application:

**`G4NEUTRONHPDATA`** [path to the G4NDL format data libraries] (mandatory).

**`/process/had/particle_hp/skip_missing_isotopes true`** This UI commands sets to zero the cross section of the isotopes which are not present in the neutron library. If GEANT4 doesn't find an isotope, then it looks for the natural composition data of that element. Only if the element is not found then the cross section is set to zero. On the contrary, if this variable is not defined, GEANT4 looks then for the neutron data of another isotope close in Z and A, which will have completely different nuclear properties and lead to incorrect results (highly recommended).

**`/process/had/particle_hp/do_not_adjust_final_state true`** Without this UI command, a GEANT4 model that attempts to satisfy the energy and momentum conservation in some nuclear reactions, by generating artificial gamma rays. By setting such a variable one avoids the correction and leads to the result obtained with the ENDF-6 libraries. Even though energy and momentum conservation are desirable, the ENDF-6 libraries do not provide the necessary correlations between secondary particles for satisfying them in all cases. On the contrary, ENDF-6 libraries intrinsically violate energy and momentum conservation for several processes and have been built for preserving the overall average quantities such as average energy releases, average number of secondaries… (highly recommended).

The G4NDL format libraries are based on the ENDF-6 format libraries, which contain evaluated (i.e. recommended) nuclear data prepared for their use in transport codes. These data are essentially nuclear reaction cross sections together with the distribution in energy and angle of the secondary reaction products. As a consequence of how the data is written in the ENDF files, there are some features that may be or may be not expected in the results of a Monte Carlo calculation.

The information concerning the creation of the reaction products can be incomplete and/or uncorrelated, in the sense that is described below:

1. **Incomplete information.**
   This applies when there is no information about how to generate a secondary particle. As an example, it is possible to have only the cross section data of an (n,p) reaction, without any information concerning the energy and angle of the secondary proton. In this case GEANT4 will produce the proton considering that it is emitted isotropically in the center of mass frame, with an energy which is deduced from assuming that the residual nucleus is in its ground state.

2. **Uncorrelated information.**
   This applies when:
   1. The energy and angle distributions of a reaction product may be uncorrelated. As a consequence, the reaction products can be generated with an unphysical energy-angle relationship.
   2. The energy-angle distributions of different reaction products of a certain reaction are always uncorrelated. As an example, consider that in a (n, 2p) reaction at a certain neutron energy both resulting protons can

be emitted with energies ranging from 0 to 5MeV. In this case the energy and angle of each proton will be sampled independently of the energy and angle of the other proton, so there will be events in which both protons will be emitted with energies close to 5 MeV and there will also be events in which both protons will be emitted with energies close to 0 MeV. As a consequence, energy and angular momentum won't be conserved event by event. However, energy will be conserved in average and the resulting proton energy spectrum will be correctly produced.

3. **Concatenated reactions.**
   There are some cases where several nuclear reactions are put together as if they were a single reaction (`MT=5` reaction, in ENDF-6 format nomenclature). In those cases the information consists in a cross section, which is the sum of all of them, plus a reaction product yield and energy-angle distributions for each secondary particle. In this case the amount of each secondary particle produced has to be sampled every time the reaction occurs, and it is done independently of the amount of the other secondary particles produced.
   Thus, in this case neither the energy and angular momentum nor the number of nucleons is conserved event by event, but all the quantities should be conserved in average. As a consequence, it is also not possible to deduce which are the residual nuclei produced, since no information is available concerning what are the specific nuclear reactions which take place. It has to be said that sometimes ENDF libraries include the residual nuclei as an outgoing particle. However, GEANT4 does not manage that information, at present. This situation is quite uncommon in neutron data libraries up to 20 MeV. However, it is quite common to find it in charged particle libraries below 20 MeV or in neutron libraries above 20 MeV.

As a consequence of what has been presented above, some general features can be expected in the results of a Monte Carlo calculation performed with the G4NeutronHP package:

- The neutron transport, which means how the neutron looses energy in the collisions, when and how it is absorbed. . . , is quite trustable, since the main purpose of the ENDF neutron libraries is to perform this neutron transport.
- The production of neutrons due to neutron induced nuclear reactions is usually trustable, with the exception of the energy-angle correlations when several neutrons are produced in the same nuclear reaction.
- The results concerning the production of charged particles have to be always questioned. A look into the ENDF format library used can indicate which results are trustable and which are not. This can be done, for example, in t2 web-page, among other websites.
- The results concerning the production of $\gamma$-rays have to be questioned always. For example, the information on the number and energies of $\gamma$-rays emitted in the neutron capture process is incomplete for almost all the nuclei and is frequently also uncorrelated. When the information is available, it will be used, but one can obtain results which are quite far from reality on an event by event basis: the total energy of the cascade won't be correct in many cases and only some specific $\gamma$-rays which are stored in the neutron databases will be emitted. If there isn't any information concerning these $\gamma$-rays, GEANT4 will use a simple a model instead which is generally missing the relevant spectroscopic information. The results concerning the generation of residual nuclei (for example, in activation calculations) are usually trustable, with the exception of libraries with MT=5 reactions, as described above (*uncorrelated*).

As a general conclusion, users should always be critical with the results obtained with Monte Carlo simulation codes, and this also applies to GEANT4. They have to anticipate which results can be trusted and which results should be questioned. For the particular case of the a closer look into the underlying evaluated nuclear data in the ENDF format libraries will allow to check what is the information available in a certain library for some specific isotope and a certain reaction. There are several public nuclear data sites like t2 web.

The transport of very low energy neutrons (below 5 eV) has to be performed using the thermal neutron data libraries. At these energies, the fact that the nuclei are in atoms which form part of a certain molecule inside a material (crystal lattice, liquid, plastic. . . ) plays an important role, since there can be a transference of momentum between the neutron and the whole structure of the material, not only with the nucleus. This is of particular importance for material used as neutron moderators, i.e., materials with low A (mass number) used to decrease the incident neutron energy in only a few collisions. Since the property is related to the nucleus in the material, as an example, there is the need for having different thermal libraries for Hydrogen in polyethylene, Hydrogen in water and so on.

If neutron collisions at these energies are relevant for the problem to be simulated, thermal libraries should be used for the materials if they are available. If they are not, the results obtained from the simulation will not be trustable in the neutron energy range below 5 eV, especially when using low mass elements in the simulation.

To use the thermal libraries the following lines should be included in the physics list:

```
G4HadronElasticProcess* theNeutronElasticProcess = new G4HadronElasticProcess;
// Cross Section Data set
G4NeutronHPElasticData* theHPElasticData = new G4NeutronHPElasticData;
theNeutronElasticProcess->AddDataSet(theHPElasticData);
G4NeutronHPThermalScatteringData* theHPThermalScatteringData = new␣
↪G4NeutronHPThermalScatteringData;
theNeutronElasticProcess->AddDataSet(theHPThermalScatteringData);
// Models
G4NeutronHPElastic* theNeutronElasticModel = new G4NeutronHPElastic;
theNeutronElasticModel->SetMinEnergy(4.0*eV);
theNeutronElasticProcess->RegisterMe(theNeutronElasticModel);
G4NeutronHPThermalScattering* theNeutronThermalElasticModel = new G4NeutronHPThermalScattering;
theNeutronThermalElasticModel->SetMaxEnergy(4.0*eV);
theNeutronElasticProcess->RegisterMe(theNeutronThermalElasticModel);
// Apply Processes to Process Manager of Neutron
G4ProcessManager* pmanager = G4Neutron::Neutron()->GetProcessManager();
pmanager->AddDiscreteProcess(theNeutronElasticProcess);
```

And the materials should be defined with a specific name. For example, to use the thermal library for Hydrogen in water, the water should be defined as:

```
G4Element* elTSHW = new G4Element("TS_H_of_Water", "H_WATER", 1.0, 1.0079*g/mole);
G4Material* matH2O_TS = new G4Material("Water_TS", density=1.0*g/cm3, ncomponents=2);
matH2O_TS->AddElement(elTSHW,natoms=2);
matH2O_TS->AddElement(elO,natoms=1);
```

where the important thing is the name **"TS_H_of_Water"**, which is a specific name used by G4NeutronHP. In order to see which thermal libraries are available, they can be found in the G4NDL4.0/ThermalScattering folder (or equivalent, for other neutron libraries). Then, one has to look into the `G4NeutronHPThermalScatteringNames.cc` source file, under `source/processes/hadronic/models/neutron_hp/src`. There are some lines similar to:

```
names.insert(std::pair<G4String,G4String>("TS_H_of_Water", "h_water"));
```

where **"TS_H_of_Water"** means Hydrogen in water. Names similar to **"TS_H_of_Water"** like **"TS_C_of_Graphite"** or **"TS_H_of_Polyethylene"** can be found and used in the same way as described above.

### High-precision charged particle interactions (ParticleHP)

By default in ParticleHP the final state is adjusted to ensure better conservation laws (for charge, energy, momentum, baryon number). However, projectile charged particles can be improved by using the following UI command: `/process/had/particle_hp/do_not_adjust_final_state true`

The adjustment of the final state is recommended for realistic detector response in the case of neutron interactions. For the use-case of reactor physics and dosimetry, where *average* quantities are important, not adjusting the final state (i.e. setting the above environment variable) improves accuracy.

Note that, for the time being, the UI command `/process/had/particle_hp/do_not_adjust_final_state true` affects both primary neutrons and charged particles, so be careful which is the use-case and observable quantity you are interested in.

### Switching statistical nuclear de-excitation models

Nuclear reactions at intermediate energies (from a few MeV to a few GeV) are typically modelled in two stages. The first, fast reaction stage is described by a dynamical model (quantum molecular dynamics, intranuclear cascade, pre-compound, etc.) and often results in the production of one or several excited nuclei. The second reaction stage describes the de-excitation of the excited nuclei and it is usually handled by statistical de-excitation models. The models for the two reaction stages can in principle be chosen independently, but the current design of the GEANT4 hadronics framework makes it difficult to do this at the physics-list level. However, another solution exists.

GEANT4 provides several nuclear de-excitation modules. The default one is `G4ExcitationHandler`, which is described in detail in the Physics Reference Manual. The Bertini-style `G4CascadeInterface` uses an internal de-excitation model. The *ABLA V3* model is also available.

**Options** are available for steering of the pre-compound model and the de-excitation module. These options may be invoked by the new C++ interface class `G4DeexPrecoParameters`. The interface `G4NuclearLevelData::Instance()->GetParameters()` is thread safe, parameters are shared between threads, and parameters are shared between all de-excitation and pre-compound classes. Parameters may be modified at G4State_PreInit state of GEANT4. This class has the following public methods:

- Dump()
- StreamInfo(std::ostream&)
- SetLevelDensity(G4double)
- SetR0(G4double)
- SetTransitionsR0(G4double)
- SetFBUEnergyLimit(G4double)
- SetFermiEnergy(G4double)
- SetPrecoLowEnergy(G4double)
- SetPrecoHighEnergy(G4double)
- SetPhenoFactor(G4double)
- SetMinExcitation(G4double)
- SetMaxLifeTime(G4double)
- SetMinExPerNucleounForMF(G4double)
- SetMinEForMultiFrag(G4double)
- SetMinZForPreco(G4int)
- SetMinAForPreco(G4int)
- SetPrecoModelType(G4int)
- SetDeexModelType(G4int)
- SetTwoJMAX(G4int)
- SetVerbose(G4int)
- SetNeverGoBack(G4bool)
- SetUseSoftCutoff(G4bool)
- SetUseCEM(G4bool)
- SetUseGNASH(G4bool)
- SetUseHETC(G4bool)
- SetUseAngularGen(G4bool)
- SetPrecoDummy(G4bool)
- SetCorrelatedGamma(G4bool)
- SetStoreICLevelData(G4bool)
- SetInternalConversionFlag(G4bool)
- SetLevelDensityFlag(G4bool)
- SetDiscreteExcitationFlag(G4bool)
- SetIsomerProduction(G4bool)
- SetDeexChannelType(G4DeexChannelType)

It is possible to replace the default de-excitation model with *ABLA V3* for any intranuclear-cascade model in GEANT4 except `G4CascadeInterface`. The easiest way to do this is to call the `SetDeExcitation()` method of the relevant intranuclear-cascade-model interface. This can be done even if you are using one of the reference physics lists. The technique is the following.

For clarity's sake, assume you are using the *FTFP_INCLXX* physics list, which uses *INCL++*, the Liege Intranuclear Cascade model (`G4INCLXXInterface`) at intermediate energies. You can couple *INCL++* to *ABLA V3* by adding a run action (*Usage of User Actions*) and adding the following code snippet to `BeginOfRunAction()`.

Listing 5.1: Coupling the INCL++ model to ABLA V3

```
#include "G4HadronicInteraction.hh"
#include "G4HadronicInteractionRegistry.hh"
#include "G4INCLXXInterface.hh"
#include "G4AblaInterface.hh"

void MyRunAction::BeginOfRunAction(const G4Run*)
{
  // Get hold of pointers to the INCL++ model interfaces
  std::vector<G4HadronicInteraction *> interactions = G4HadronicInteractionRegistry::Instance()
    ->FindAllModels(G4INCLXXInterfaceStore::GetInstance()->getINCLXXVersionName());
  for(std::vector<G4HadronicInteraction *>::const_iterator iInter=interactions.begin(),␣
↪e=interactions.end();
      iInter!=e; ++iInter) {
    G4INCLXXInterface *theINCLInterface = static_cast<G4INCLXXInterface*>(*iInter);
    if(theINCLInterface) {
      // Instantiate the ABLA model
      G4HadronicInteraction *interaction = G4HadronicInteractionRegistry::Instance()->FindModel(
↪"ABLA");
      G4AblaInterface *theAblaInterface = static_cast<G4AblaInterface*>(interaction);
      if(!theAblaInterface)
        theAblaInterface = new G4AblaInterface;
      // Couple INCL++ to ABLA
      G4cout << "Coupling INCLXX to ABLA" << G4endl;
      theINCLInterface->SetDeExcitation(theAblaInterface);
    }
  }
}
```

This technique may be applied to any intranuclear-cascade model (i.e. models that inherit from `G4VIntraNuclearTransportModel`), except `G4CascadeInterface`. For example, if your physics list relies on the *Binary-Cascade* model (e.g. *FTF_BIC*), you'll need to do

```
// Get hold of a pointer to the Binary-Cascade model interface
std::vector<G4HadronicInteraction *> interactions = G4HadronicInteractionRegistry::Instance()
  ->FindAllModels("Binary Cascade");
for(std::vector<G4HadronicInteraction *>::const_iterator iInter=interactions.begin(),␣
↪e=interactions.end();
    iInter!=e; ++iInter) {
  G4BinaryCascade *theBICInterface = static_cast<G4BinaryCascade*>(*iInter);
  if(theBICInterface) {

    // Instantiate ABLA V3 as in the example above
    // [...]

    // Couple BIC to ABLA
    theBICInterface->SetDeExcitation(theAblaInterface);
  }
}
```

### 5.2.4 Particle Decay Process

This section briefly introduces decay processes installed in GEANT4. For details of the implementation of particle decays, please refer to the Physics Reference Manual.

#### Particle Decay Class

GEANT4 provides a `G4Decay` class for both `at rest` and `in flight` particle decays. `G4Decay` can be applied to all particles except:

**massless particles, i.e.,** `G4ParticleDefinition::thePDGMass <= 0`
**particles with "negative" life time, i.e.,** `G4ParticleDefinition::thePDGLifeTime < 0`
**shortlived particles, i.e.,** `G4ParticleDefinition::fShortLivedFlag = True`

Decay for some particles may be switched on or off by using `G4ParticleDefinition::SetPDGStable()` as well as `ActivateProcess()` and `InActivateProcess()` methods of `G4ProcessManager`.

`G4Decay` proposes the step length (or step time for `AtRest`) according to the lifetime of the particle unless `PreAssignedDecayProperTime` is defined in `G4DynamicParticle`.

The `G4Decay` class itself does not define decay modes of the particle. GEANT4 provides two ways of doing this:

- using `G4DecayChannel` in `G4DecayTable`, and
- using `thePreAssignedDecayProducts` of `G4DynamicParticle`

The `G4Decay` class calculates the `PhysicalInteractionLength` and boosts decay products created by `G4VDecayChannel` or event generators. See below for information on the determination of the decay modes.

An object of `G4Decay` can be shared by particles. Registration of the decay process to particles in the `ConstructPhysics` method of *PhysicsList* (see *How to Specify Physics Processes*) is shown in Listing 5.2.

Listing 5.2: Registration of the decay process to particles in the `ConstructPhysics` method of *PhysicsList*.

```
#include "G4Decay.hh"
void MyPhysicsList::ConstructGeneral()
{
  // Add Decay Process
  G4Decay* theDecayProcess = new G4Decay();
  theParticleIterator->reset();
  while( (*theParticleIterator)() ){
    G4ParticleDefinition* particle = theParticleIterator->value();
    G4ProcessManager* pmanager = particle->GetProcessManager();
    if (theDecayProcess->IsApplicable(*particle)) {
      pmanager ->AddProcess(theDecayProcess);
      // set ordering for PostStepDoIt and AtRestDoIt
      pmanager ->SetProcessOrdering(theDecayProcess, idxPostStep);
      pmanager ->SetProcessOrdering(theDecayProcess, idxAtRest);
    }
  }
}
```

## Decay Table

Each particle has its `G4DecayTable`, which stores information on the decay modes of the particle. Each decay mode, with its branching ratio, corresponds to an object of various “decay channel” classes derived from `G4VDecayChannel`. Default decay modes are created in the constructors of particle classes. For example, the decay table of the neutral pion has `G4PhaseSpaceDecayChannel` and `G4DalitzDecayChannel` as follows:

```
// create a decay channel
G4VDecayChannel* mode;
// pi0 -> gamma + gamma
mode = new G4PhaseSpaceDecayChannel("pi0",0.988,2,"gamma","gamma");
table->Insert(mode);
// pi0 -> gamma + e+ + e-
mode = new G4DalitzDecayChannel("pi0",0.012,"e-","e+");
table->Insert(mode);
```

Decay modes and branching ratios defined in GEANT4 are listed in *Definition of a particle*.

Branching ratios and life time can be set in tracking time.

```
// set lifetime
G4Neutron::Neutron()->SetPDGLifeTime(885.7*second);
// allow neutron decay
G4Neutron::Neutron()->SetPDGStable(false);
```

Branching ratios and life time can be modified by using user commands, also.

**Example: Set 100% br for dalitz decay of pi0**

```
Idle> /particle/select pi0
Idle> /particle/property/decay/select 0
Idle> /particle/property/decay/br 0
Idle> /particle/property/decay/select 1
Idle> /particle/property/decay/br 1
Idle> /particle/property/decay/dump
      G4DecayTable:  pi0
         0:  BR:  0  [Phase Space]   :   gamma gamma
         1:  BR:  1  [Dalitz Decay]  :   gamma e- e+
```

### Pre-assigned Decay Modes by Event Generators

Decays of heavy flavor particles such as B mesons are very complex, with many varieties of decay modes and decay mechanisms. There are many models for heavy particle decay provided by various event generators and it is impossible to define all the decay modes of heavy particles by using `G4VDecayChannel`. In other words, decays of heavy particles cannot be defined by the GEANT4 decay process, but should be defined by event generators or other external packages. GEANT4 provides two ways to do this: `pre-assigned decay mode` and `external decayer`.

In the latter approach, the class `G4VExtDecayer` is used for the interface to an external package which defines decay modes for a particle. If an instance of `G4VExtDecayer` is attached to `G4Decay`, daughter particles will be generated by the external decay handler.

In the former case, decays of heavy particles are simulated by an event generator and the primary event contains the decay information. `G4VPrimaryGenerator` automatically attaches any daughter particles to the parent particle as the PreAssignedDecayProducts member of `G4DynamicParticle`. `G4Decay` adopts these pre-assigned daughter particles instead of asking `G4VDecayChannel` to generate decay products.

In addition, the user may assign a `pre-assigned` decay time for a specific track in its rest frame (i.e. decay time is defined in the proper time) by using the `G4PrimaryParticle::SetProperTime()` method. `G4VPrimaryGenerator` sets the PreAssignedDecayProperTime member of `G4DynamicParticle`. `G4Decay` uses this decay time instead of the life time of the particle type.

## 5.2.5 Note on the time threshold for radioactive decay of ions

Since Geant4 version 11.0, a time threshold for the radioactive decay of ions has been introduced: nuclides with a sampled lifetime longer than this threshold are ignored (i.e. killed with neither daughters nor deposited energy). This is aimed to avoid confusing results in applications where a time window is not explicitly defined, and the effect of very slow radioactive decays of ions is assumed to be negligible - which is not always true!

In Geant4 versions 11.0 and 11.1 (and subsequent patches), the default time threshold was $10^{27}$ns, corresponding to about twice the age of the universe. Starting with Geant4 version 11.2, the default time threshold has been changed to 1 year.

The motivation was the same as that which brought us to introduce the threshold in Geant4 version 11.0, i.e. to avoid unexpected differences in the energy depositions in thick set-ups (e.g. calorimeters or shielding structures) between physics lists without Radioactive Decay (e.g. `FTFP_BERT`, `QGSP_BIC`, `QBBC`, etc.) and those with it (e.g. `FTFP_BERT_HP`, `QGSP_BIC_HP`, `Shielding`, etc.), when a time window is not explicitly defined by users. Moreover, in these cases, the results are also depending on the value of the range threshold for proton (because target nuclei which receive an elastic recoil above a certain kinetic energy become tracks, and therefore can have radioactive decays).

For applications where radioactive decays of ions do play an important role, it is recommended to increase the default time threshold of these decays to a very high value, e.g. 1.0e+60 years.

This can be done in one of the following three ways:

1. Via UI command, e.g.
   ```
   /process/had/rdm/thresholdForVeryLongDecayTime 1.0e+60 year
   ```
   (command to be used after `/run/initialization`)
2. Via C++ interface, e.g.
   ```
   G4HadronicParameters::Instance()->SetTimeThresholdForRadioactiveDecay(
   1.0e+60*CLHEP::year )
   ```
   (to be placed in your main program before **run initialization**)
3. Via the second parameter of the constructor of the class:
   `G4RadioactiveDecay` (for analogue mode only) or
   `G4Radioactivation` (for both analogue or biased mode), e.g.
   ```
   G4RadioactiveDecay( "RadioactiveDecay", 1.0e+60*CLHEP::year )
   ```

or `G4Radioactivation( "Radioactivation", 1.0e+60*CLHEP::year )` (this is for custom physics lists, before run initialization).

In the examples, we have followed the first method.

## 5.2.6 Gamma-nuclear and Lepto-nuclear Processes

Gamma-nuclear and lepto-nuclear reactions are handled in GEANT4 as hybrid processes which typically require both electromagnetic and hadronic models for their implementation. While neutrino-induced reactions are not currently provided, the GEANT4 hadronic framework is general enough to include their future implementation as a hybrid of weak and hadronic models.

The general scheme followed is to factor the full interaction into an electromagnetic (or weak) vertex, in which a virtual particle is generated, and a hadronic vertex in which the virtual particle interacts with a target nucleus. In most cases the hadronic vertex is implemented by an existing GEANT4 model which handles the intra-nuclear propagation.

The cross sections for these processes are parameterizations, either directly of data or of theoretical distributions determined from the integration of lepton-nucleon cross sections double differential in energy loss and momentum transfer.

Electro-nuclear reactions in GEANT4 are handled by the classes `G4ElectronNuclearProcess` and `G4PositronNuclearProcess`, which are both implmented by `G4ElectroVDNuclearModel`. This model consists of three sub-models: code which generates the virtual photon from the lepton-nucleus vertex, the Bertini-style cascade to handle the low and medium energy photons, and the FTFP model to handle the high energy photons.

Muon-nuclear reactions are handled similarly. The process `G4MuonNuclearProcess` can be assigned the `G4MuonVDNuclearModel` which in turn is implemented by three sub-models: virtual gamma generation code, Bertini-style cascade and the FTFP model.

### Resonance effects

Note that the model that generates secondary particles does not take resonance effects into account. This may be particularly important in the region of the giant dipole resonance, at ~20MeV for high-Z materials (an important region for shielding in some medical applications.) Here, about 2/3 of photonuclear interactions may not produce neutrons.

In order to obtain correct photonuclear production in this regime, the user should use a LEND, data-driven model. Alternatively, if the environment variable `G4CASCADE_CHECK_PHOTONUCLEAR` is set, the model will ensure that, for incident energies less than 50 MeV, the mass of the nucleus changes during the interaction.

## 5.2.7 Optical Photon Processes

A photon is considered to be *optical* when its wavelength is much greater than the typical atomic spacing. In GEANT4 optical photons are treated as a class of particle distinct from their *gamma* cousins. Optical photons have different processes than gamma particles. An important use case for optical photons is that they interact with boundaries between volumes, undergoing reflection, refraction, etc.

**Note:** There is no transition between the optical photon and gamma particle classes: a *gamma* will never become an *optical photon*.

Optical photons are generated in GEANT4 by

- Cerenkov effect (class `G4Cerenkov`)
- Scintillation (class `G4Scintillation`)

The source code for these classes is in the `source/processes/electromagnetic/xrays` directory.

Optical photons interact through the processes

- Absorption (class `G4OpAbsorption`)
- Rayleigh scattering (class `G4OpRayleigh`)
- Mie scattering (class `G4OpMieHG`)
- Wave-length shifting (classes `G4OpWLS` and `G4OpWLS2`)
- Boundary scattering (class `G4OpBoundary`)

The source code for these classes is in the `source/processes/optical` directory.

Optical photons are generated in GEANT4 without regard to energy conservation and their energy must therefore not be tallied as part of the energy balance of an event.

The are several steps to simulate optical photons in GEANT4.

- The optical photon and optical processes must be defined and configured
- Optical properties need to be assigned to relevant materials and surfaces
- If an optical photon is a primary particle, its polarization should be set

### Defining optical processes: `G4OpticalPhysics` constructor

The most straightforward way of using optical physics is to use the `G4OpticalPhysics` constructor in main(), as in the extended optical examples. This automatically includes all the optical physics processes and provides a default configuration. The configuration is stored in the class `G4OpticalParameters`, and can be accessed via the class `G4OpticalParametersMessenger`. Both macro commands and C++ methods are available.

An example of using `G4OpticalPhysics` is shown in Listing 5.3.

Listing 5.3: An example of using the `G4OpticalPhysics` constructor in main().

```
#include "G4OpticalPhysics.hh"

...

G4VModularPhysicsList* physicsList = new FTFP_BERT;  // for example
G4OpticalPhysics* opticalPhysics = new G4OpticalPhysics();

physicsList->RegisterPhysics(opticalPhysics);
runManager->SetUserInitialization(physicsList);

// to set parameters in code, if wanted
auto opticalParams = G4OpticalParameters::Instance();
opticalParams->SetWLSTimeProfile("delta");
```

See the sections for each process for details on process-specific commands in the class `G4OpticalParametersMessenger`. There are two general commands:

1. `/process/optical/verbose int` sets the verbosity of all processes to the given value. 0 is silent, 1 is printing during initialization only, 2 is printing during tracking
2. `/process/optical/processActivation name bool` is used to deactivate individual processes. `name` may be one of Cerenkov, Scintillation, OpAbsorption, OpRayleigh, OpMieHG, OpBoundary, OpWLS, OWLS2. By default, all the processes are activated.

The optical parameters can be printed by invoking `G4OpticalParameters::Instance()->Dump()`.

**Note:** In version 10.7, redundant commands were marked as deprecated, but were still available. In version 11, these commands have been removed.

### Setting the polarization

For the simulation of optical photons to work correctly in GEANT4, they must have a linear polarization. This is unlike most other particles in GEANT4 but is automatically and correctly done for optical photons that are generated as secondaries by existing processes in GEANT4. If the user wishes to start optical photons as primary particles, they must set the linear polarization using particle gun methods, the General Particle Source, or their PrimaryGeneratorAction. For an unpolarized source, the linear polarization should be sampled randomly for each new primary photon. See the extended optical examples for methods to set the polarization.

### Defining material properties

The optical properties of the medium which are key to the implementation of these types of processes are stored as entries in a `G4MaterialPropertiesTable` which is a private data member of the `G4Material` class.

Each entry in the `G4MaterialPropertiesTable` consists of a *key* and *value* pair. The key is used to retrieve the corresponding value. Keys are defined in two `enums` (and thus are `G4ints`). Keys are also available as `G4Strings`. Properties are added using the `G4String` key, but may be accessed by either the `G4String` key or the `G4int` key. Material property names are listed in tables for each process, below.

These properties may be independent of photon energy (denoted “Constant” or “Const”) or they may be expressed as a function of the photon’s energy. In the case of Constant parameters, the value is a `G4double`. Methods to access the Constant parameters have “Const” in their names.

In the case of energy-dependent properties, the value is a `G4MaterialPropertyVector` (which is a typedef to `G4PhysicsFreeVector`). Energy dependent properties may be added by specifying a `G4MaterialPropertyVector`, or by passing two std::vectors of type `G4double`. The first vector is the energy and the second vector is the property value at that energy. The numbers of elements in the two vectors must be the same (if std::vectors are used, GEANT4 will check that the numbers of elements in the two vectors are the same).

For backwards compatibility, energy dependent properties may also be created by specifying two C arrays of doubles. In this case, it is up to the user to ensure that both arrays have the same number of values, and the number of values must be passed as an argument.

Typically, a user only needs to add properties. Authors of processes that need these properties will need to access the values. Methods to add properties are:

- `void AddConstProperty(const G4String& key, G4double PropertyValue)`
- `void AddConstProperty(const char* key, G4double PropertyValue)`
- `G4MaterialPropertyVector* AddProperty(const G4String& key, const std::vector<G4double>& photonEnergies, const std::vector<G4double>& propertyValues, G4bool createNewKey = false, G4bool spline = false)`
- `G4MaterialPropertyVector* AddProperty(const char* key, G4double* PhotonEnergies, G4double* PropertyValues, G4int NumEntries, G4bool createNewKey = false, G4bool spline = false)`
- `void AddProperty(const G4String& key, G4MaterialPropertyVector* opv, G4bool createNewKey = false)`
- `void AddProperty(const char* key, G4MaterialPropertyVector* opv, G4bool createNewKey = false)`
- `void AddProperty(const G4String& key, const G4String& mat)`

The createNewKey argument, if `false`, will check that the key string is one of the default keys. See *User-defined properties* for more information. The spline argument enables spline interpolation of the data.

An example of adding material properties to a material is shown in Listing 5.4. In this example the interpolation of the `G4MaterialPropertyVector` is to be done by a spline fit. The default is a linear interpolation.

Listing 5.4: Example of optical properties added to a `G4MaterialPropertiesTable` and linked to a `G4Material`

```
G4Material* scintillator = new G4Material(/*...*/);

std::vector<G4double> energy     = {2.034*eV, 3.*eV, 4.136*eV};
std::vector<G4double> rindex     = {1.3435, 1.351, 1.3608};
std::vector<G4double> absorption = {344.8*cm, 850.*cm, 1450.0*cm};

G4MaterialPropertiesTable* MPT = new G4MaterialPropertiesTable();

// property independent of energy
MPT->AddConstProperty("SCINTILLATIONYIELD", 100./MeV);

// properties that depend on energy
MPT->AddProperty("RINDEX", energy, rindex);
MPT->AddProperty("ABSLENGTH", energy, absorption);

scintillator->SetMaterialPropertiesTable(MPT);
```

**Note:** Starting in version 11.0, GEANT4 will check that the property name is in the list of pre-defined properties. This avoids errors due to spelling mistakes. If you want to define a new property name, see *User-defined properties*.

**Note:** Digitized data of refractive indices for many materials can be accessed for instance at https://refractiveindex.info.

## Pre-defined properties

Starting in version 11.0, some optical material properties are defined in GEANT4. Currently, these are the refractive indices for "Air", "Water", "PMMA", and "Fused Silica". The precise values are found in `source/materials/include/G4OpticalMaterialProperties.hh`.

To use them, add them to the material properties table as follows:

```
G4MaterialPropertiesTable* MT = new G4MaterialPropertiesTable();
MT->AddProperty("RINDEX", "Fused Silica");
```

## User-defined properties

One may create their own properties, for example, for use in custom processes. To facilitate this, the various `AddProperty()`/`AddConstProperty()` methods have a default argument `createNewKey`. Set this to `true` to allow a new key name.

```
myMPT->AddConstProperty("USERDEFINEDCONST", 3.14, true);
```

This value may accessed by the string name:

```
G4double val = myMPT->GetConstProperty("USERDEFINEDCONST");
```

but there is a potential speed-up. Material properties are stored internally in a vector. To access a material property, first find the index of the property (e.g. at initialization), then use the index during the event loop.

```
G4int index = myMPT->GetConstPropertyIndex("USERDEFINEDCONST");
...
G4double val = myMPT->GetConstProperty(index);
```

It is possible to test if a property has been defined. In the case of constant properties, the methods `ConstPropertyExists(const G4String& key)`, `ConstPropertyExists(const char* key)`, `ConstPropertyExists(const G4int index)` return `true` if the property is defined, false otherwise. For energy-dependent properties, the `GetProperty(...)` methods return `nullptr` if the property has not been defined.

### Cerenkov Effect

The radiation of Cerenkov light occurs when a charged particle moves through a dispersive medium faster than the group velocity of light in that medium. Photons are emitted on the surface of a cone, whose opening angle with respect to the particle's instantaneous direction decreases as the particle slows down. At the same time, the frequency of the photons emitted increases, and the number produced decreases. When the particle velocity drops below the local speed of light, the radiation ceases and the emission cone angle collapses to zero. The photons produced by this process have an inherent polarization perpendicular to the cone's surface at production.

To generate Cerenkov optical photons in a material, refractive index must be specified using the material property name `RINDEX`.

The flux, spectrum, polarization and emission of Cerenkov radiation in the `AlongStepDoIt` method of the class `G4Cerenkov` follow well-known formulae, with two inherent computational limitations. The first arises from step-wise simulation, and the second comes from the requirement that numerical integration calculate the average number of Cerenkov photons per step. The process makes use of a `G4PhysicsTable` which contains incremental integrals to expedite this calculation.

The time and position of Cerenkov photon emission are calculated from quantities known at the beginning of a charged particle's step. The step is assumed to be rectilinear even in the presence of a magnetic field. The user may limit the step size by specifying a maximum (average) number of Cerenkov photons created during the step, using the `setMaxPhotons` command. The actual number generated will necessarily be different due to the Poissonian nature of the production. In the present implementation, the production density of photons is distributed evenly along the particle's track segment, even if the particle has slowed significantly during the step. The step can also be limited with the `setMaxBetaChange` command, where the argument is the allowed change in percent.

The large number of optical photons that can be produced (about 300/cm in water) can fill the available memory. GEANT4 by default will track the Cerenkov photons produced in a step before continuing to track the primary particle. This may be changed using the `setTrackSecondariesFirst` command.

### Configuration

Material property names used in the process are given in the following table.

Table 5.2: Material properties for the Cerenkov process.

| Name | Type | Description | Unit Category |
|---|---|---|---|
| RINDEX | Energy-dependent | Refractive index | Unitless |

These parameters are available to configure the process.

- Set the step size to limit the number of photons produced (on average) to a given value (an integer N)
  - macro command: /process/optical/cerenkov/setMaxPhotons N
  - C++ statement: G4OpticalParameters::Instance()->SetMaxNumPhotonsPerStep(G4int);
  - default value: 100

- Set the maximum change in $\beta = v/c$ in a step, expressed in percent.
  - macro command: /process/optical/cerenkov/setMaxBetaChange X.X
  - C++ statement: G4OpticalParameters::Instance()->SetMaxBetaChangePerStep(G4double);
  - default value: 10.0
- Specify whether to add Cerenkov photons to the stack, and track them.
  - macro command: /process/optical/cerenkov/setStackPhotons true
  - C++ statement: G4OpticalParameters::Instance()->SetCerenkovStackPhotons(G4bool);
  - default value: true
- Specify whether to track secondaries produced in the step before continuing with primary.
  - macro command: /process/optical/cerenkov/setTrackSecondariesFirst true
  - C++ statement: G4OpticalParameters::Instance()->SetCerenkovTrackSecondariesFirst(G4bool);
  - default value: true
- Set the verbosity of the process. 0 = silent; 1 = initialisation; 2 = during tracking
  - macro command: /process/optical/cerenkov/verbose
  - C++ statement: G4OpticalParameters::Instance()->SetCerenkovVerbosity(G4int);
  - default value: 1

### Scintillation

A scintillating material is characterised by the number of optical photons produced (the yield), their spectrum, and the distribution of emission times. The yield may be dependent on the type of primary particle. The distribution of emission times can be characterised by an exponential rise time, and a decay with one or more time constants. Starting with GEANT4 version 10.7, it is possible to specify up to three decay time constants, for particle-independent and particle-dependent yields. This new method uses different material property names than used in previous versions. The method used in previous versions is no longer available in GEANT4 version 11.0. All of the capabilities of the previous method are available with the new method.

The scintillation yield may depend on the type of primary particle. In this case, the material parameter names are modified from the case where the yield is independent of primary particle.

Each photon's frequency is sampled from the empirical spectrum. The photons originate evenly along the track segment and are emitted uniformly into the $4\pi$ solid angle with a random linear polarization perpendicular to their momentum direction, and an emission time characteristic for the scintillation component.

**Important:** Starting in GEANT4 version 11.0, only the new method (called "enhanced" in version 10.7) is available to specify decay time constants. It is no longer necessary to set the optical parameter `setEnhancedTimeConstants` to true.

#### Scintillation independent of particle type

If the scintillation yield is independent of particle type, the yield is specified using the material constant property `SCINTILLATIONYIELD`. The mean number of optical photons in a step is calculated as the `SCINTILLATIONYIELD` property, times the yield factor, times either the `VisibleEnergyDepositAtAStep` (if Birks' saturation is used) or the total energy deposit (otherwise). (Note the yield factor is redundant here and should not be used. See the description of the old scintillation methods for its use.) The property `SCINTILLATIONYIELD` is expressed in number per energy.

If the calculated mean number of optical photons for the step is less than or equal to 10, the actual number is determined from a Poisson distribution with that mean, then converted to an integer. If the mean number of photons is greater than 10, the number of photons is chosen from a Gaussian distribution with that mean, and a sigma equal to the square root of the the mean times a factor called the resolution scale. This factor is set using the material constant property `RESOLUTIONSCALE`. A resolution scale of zero produces no fluctuation.

There may be 1 to 3 decay component with independent spectra and rise and decay time constants.

If there is one component, specify the decay time constant with the material constant property `SCINTILLATIONTIMECONSTANT1`. If a non-zero rise time is wanted, set the optical parameter `setFiniteRiseTime` to true, and set the material constant property `SCINTILLATIONRISETIME1` to the desired value. The creation time of the photon is chosen from a distribution with these characteristics.

The energy spectrum of the emitted photons is specified using the energy-dependent material property `SCINTILLATIONCOMPONENT1`.

If there are 2 or 3 decay component, the time constant, rise time, and spectrum must be specified for the additional component. The property names are the same as for the first component, with the "1" at the end of the property name replaced with "2" or "3". Additionally, the fraction of photons in each component must be specified. Set the material constant properties `SCINTILLATIONYIELD1`, `SCINTILLATIONYIELD2`, and `SCINTILLATIONYIELD3` (if there are three components) to the relative amount of photons produced in each component. The values will be automatically normalized. The spectra are specified using the material properties `SCINTILLATIONCOMPONENT2` and `SCINTILLATIONCOMPONENT3`. The values of the time constants do not need to be ordered (i.e., `SCINTILLATIONTIMECONSTANT2` can be greater or less than `SCINTILLATIONTIMECONSTANT1`).

## Scintillation dependent on particle type

The scintillation yield, and the strength of each component, may depend on particle type. If so, set the optical parameter `setByParticleType` true, and specify a yield vector for each particle type using material property names such as `ALPHASCINTILLATIONYIELD`. The mean number of photons produced in the step is calculated as the difference in the value of the yield vector at the pre-step kinetic energy of the primary particle, and the value of the yield vector at the pre-step kinetic energy minus the energy deposit in the step:

```
ScintillationYield = yieldVector->Value(PreStepKineticEnergy)
 - yieldVector->Value(PreStepKineticEnergy - StepEnergyDeposit);
```

The relative amounts of photons produced in each component are specified using the material constant properties with key names such as `ALPHASCINTILLATIONYIELD1`, etc. If there is only one component for a primary particle, it is not necessary to specify the per-component yield e.g. `ALPHASCINTILLATIONYIELD1`.

**Note:** Starting in GEANT4 version 11.2, it is possible to specify different decay time constants for different particles. Existing code will continue to work without change.

The decay time constants may either be the same for all particles, or specified for particular particles. In order for the time constants to be different for different particles, material constant properties such as `PROTONSCINTILLATIONTIMECONSTANT1` need to be specified. For any given particle, if this property is not specified, the value specified by `SCINTILLATIONTIMECONSTANT1` is used (and similarly for channels and 3). In this way, existing code will run unchanged.

The rise time constants and the emission spectra are not dependent on particle type. These are specified in the same way as for the scintillation yield independent of particle type. `RESOLUTIONSCALE` also is equivalent.

Listing 5.5: Specification of scintillation properties in `DetectorConstruction` using enhanced time constants (from extended example LXe).

```
//Liquid Xenon
fLXe = new G4Material("LXe",z=54.,a=131.29*g/mole,density=3.020*g/cm3);

std::vector<G4double> lxe_Energy = {7.0*eV, 7.07*eV, 7.14*eV};
```

(continues on next page)

(continued from previous page)

```
std::vector<G4double> lxe_SCINT = {0.1, 1.0, 0.1};
std::vector<G4double> lxe_RIND  = {1.59, 1.57, 1.54};
std::vector<G4double> lxe_ABSL  = {35.*cm, 35.*cm, 35.*cm};
fLXe_mt = new G4MaterialPropertiesTable();
fLXe_mt->AddProperty("SCINTILLATIONCOMPONENT1", lxe_Energy, lxe_SCINT);
fLXe_mt->AddProperty("SCINTILLATIONCOMPONENT2", lxe_Energy, lxe_SCINT);
fLXe_mt->AddProperty("RINDEX",        lxe_Energy, lxe_RIND);
fLXe_mt->AddProperty("ABSLENGTH",     lxe_Energy, lxe_ABSL);
fLXe_mt->AddConstProperty("SCINTILLATIONYIELD", 12000./MeV);
fLXe_mt->AddConstProperty("RESOLUTIONSCALE", 1.0);
fLXe_mt->AddConstProperty("SCINTILLATIONTIMECONSTANT1", 20.*ns);
fLXe_mt->AddConstProperty("SCINTILLATIONTIMECONSTANT2", 45.*ns);
fLXe_mt->AddConstProperty("SCINTILLATIONYIELD1", 1.0);
fLXe_mt->AddConstProperty("SCINTILLATIONYIELD2", 0.0);
fLXe->SetMaterialPropertiesTable(fLXe_mt);

// Set the Birks Constant for the LXe scintillator
fLXe->GetIonisation()->SetBirksConstant(0.126*mm/MeV);
```

## Configuration

These parameters are available to configure the scintillation process.

- Enable particle-dependent yields
  - macro command: /process/optical/scintillation/setByParticleType
  - C++ statement: G4OpticalParameters::Instance()->SetScintByParticleType(G4bool val)
  - default value: false
- Record track information
  - macro command: /process/optical/scintillation/setTrackInfo
  - C++ statement: G4OpticalParameters::Instance()->SetScintTrackInfo(G4bool val)
  - default value: false
- Whether to track secondaries before resuming tracking of primary particle
  - macro command: /process/optical/scintillation/setTrackSecondariesFirst
  - C++ statement: G4OpticalParameters::Instance()->SetScintTrackSecondariesFirst(G4bool val)
  - default value: true
- Whether to use a finite rise time for the secondary emission time
  - macro command: /process/optical/scintillation/setFiniteRiseTime
  - C++ statement: G4OpticalParameters::Instance()->SetScintFiniteRiseTime(G4bool val)
  - default value: false
- Whether to add produced optical photons to the stack (the alternative is to kill them)
  - macro command: /process/optical/scintillation/setStackPhotons
  - C++ statement: G4OpticalParameters::Instance()->SetScintStackPhotons(G4bool val)
  - default value: true
- Set the verbosity level. 0 = silent, 1 = initialisation, 2 = during tracking
  - macro command: /process/optical/scintillation/verbose
  - C++ statement: G4OpticalParameters::Instance()->SetScintVerboseLevel(G4int val)
  - default value: 1

Table 5.3: Material properties for the optical scintillation process.

| Name | Type | Description | Unit category |
|---|---|---|---|
| ALPHASCINTILLATIONYIELD | Energy-dependent | Yield vector for alphas | 1/Energy |
| ALPHASCINTILLATIONYIELD1 | Constant | Relative yield of component 1 for alphas | Unitless |

continues on next page

Table 5.3 – continued from previous page

| Name | Type | Description | Unit category |
|---|---|---|---|
| ALPHASCINTILLATIONYIELD2 | Constant | Relative yield of component 2 for alphas | Unitless |
| ALPHASCINTILLATIONYIELD3 | Constant | Relative yield of component 3 for alphas | Unitless |
| DEUTERONSCINTILLATIONYIELD | Energy-dependent | Yield vector for deuterons | 1/Energy |
| DEUTERONSCINTILLATIONYIELD1 | Constant | Relative yield of component 1 for deuterons | Unitless |
| DEUTERONSCINTILLATIONYIELD2 | Constant | Relative yield of component 2 for deuterons | Unitless |
| DEUTERONSCINTILLATIONYIELD3 | Constant | Relative yield of component 3 for deuterons | Unitless |
| ELECTRONSCINTILLATIONYIELD | Energy-dependent | Yield vector for electrons | 1/Energy |
| ELECTRONSCINTILLATIONYIELD1 | Constant | Relative yield of component 1 for electrons | Unitless |
| ELECTRONSCINTILLATIONYIELD2 | Constant | Relative yield of component 2 for electrons | Unitless |
| ELECTRONSCINTILLATIONYIELD3 | Constant | Relative yield of component 3 for electrons | Unitless |
| IONSCINTILLATIONYIELD | Energy-dependent | Yield vector for ions | 1/Energy |
| IONSCINTILLATIONYIELD1 | Constant | Relative yield of component 1 for ions | Unitless |
| IONSCINTILLATIONYIELD2 | Constant | Relative yield of component 2 for ions | Unitless |
| IONSCINTILLATIONYIELD3 | Constant | Relative yield of component 3 for ions | Unitless |
| PROTONSCINTILLATIONYIELD | Energy-dependent | Yield vector for protons | 1/Energy |
| PROTONSCINTILLATIONYIELD1 | Constant | Relative yield of component 1 for protons | Unitless |
| PROTONSCINTILLATIONYIELD2 | Constant | Relative yield of component 2 for protons | Unitless |
| PROTONSCINTILLATIONYIELD3 | Constant | Relative yield of component 3 for protons | Unitless |
| RESOLUTIONSCALE | Constant | Factor to vary width of yield distribution | Unitless |
| SCINTILLATIONCOMPONENT1 | Energy-dependent | Energy spectrum for decay component 1 | Unitless |
| SCINTILLATIONCOMPONENT2 | Energy-dependent | Energy spectrum for decay component 2 | Unitless |
| SCINTILLATIONCOMPONENT3 | Energy-dependent | Energy spectrum for decay component 3 | Unitless |
| SCINTILLATIONRISETIME1 | Constant | Rise time for component 1 | Time |
| SCINTILLATIONRISETIME2 | Constant | Rise time for component 2 | Time |
| SCINTILLATIONRISETIME3 | Constant | Rise time for component 3 | Time |
| SCINTILLATIONTIMECONSTANT1 | Constant | Time constant for component 1 | Time |

continues on next page

Table 5.3 – continued from previous page

| Name | Type | Description | Unit category |
|---|---|---|---|
| SCINTILLATIONTIMECONSTANT2 | Constant | Time constant for component 2 | Time |
| SCINTILLATIONTIMECONSTANT3 | Constant | Time constant for component 3 | Time |
| SCINTILLATIONYIELD | Constant | Mean yield (number of particle produce per energy) | 1/Energy |
| SCINTILLATIONYIELD1 | Constant | Relative yield of component 1 | Unitless |
| SCINTILLATIONYIELD2 | Constant | Relative yield of component 2 | Unitless |
| SCINTILLATIONYIELD3 | Constant | Relative yield of component 3 | Unitless |
| TRITONSCINTILLATIONYIELD | Energy-dependent | Yield vector for tritons | 1/Energy |
| TRITONSCINTILLATIONYIELD1 | Constant | Relative yield of component 1 for tritons | Unitless |
| TRITONSCINTILLATIONYIELD2 | Constant | Relative yield of component 2 for tritons | Unitless |
| TRITONSCINTILLATIONYIELD3 | Constant | Relative yield of component 3 for tritons | Unitless |
| PROTONSCINTILLATIONTIMECONSTANT1 | Constant | Time constant for component 1 for protons | Time |
| PROTONSCINTILLATIONTIMECONSTANT2 | Constant | Time constant for component 2 for protons | Time |
| PROTONSCINTILLATIONTIMECONSTANT3 | Constant | Time constant for component 3 for protons | Time |
| DEUTERONSCINTILLATIONTIMECONSTANT1 | Constant | Time constant for component 1 for deuterons | Time |
| DEUTERONSCINTILLATIONTIMECONSTANT2 | Constant | Time constant for component 2 for deuterons | Time |
| DEUTERONSCINTILLATIONTIMECONSTANT3 | Constant | Time constant for component 3 for deuterons | Time |
| TRITONSCINTILLATIONTIMECONSTANT1 | Constant | Time constant for component 1 for tritons | Time |
| TRITONSCINTILLATIONTIMECONSTANT2 | Constant | Time constant for component 2 for tritons | Time |
| TRITONSCINTILLATIONTIMECONSTANT3 | Constant | Time constant for component 3 for tritons | Time |
| ALPHASCINTILLATIONTIMECONSTANT1 | Constant | Time constant for component 1 for alphas | Time |
| ALPHASCINTILLATIONTIMECONSTANT2 | Constant | Time constant for component 2 for alphas | Time |
| ALPHASCINTILLATIONTIMECONSTANT3 | Constant | Time constant for component 3 for alphas | Time |
| IONSCINTILLATIONTIMECONSTANT1 | Constant | Time constant for component 1 for ions | Time |
| IONSCINTILLATIONTIMECONSTANT2 | Constant | Time constant for component 2 for ions | Time |
| IONSCINTILLATIONTIMECONSTANT3 | Constant | Time constant for component 3 for ions | Time |

continues on next page

Table 5.3 – continued from previous page

| Name | Type | Description | Unit category |
|---|---|---|---|
| ELECTRONSCINTILLATIONTIMECONSTANT1 | Constant | Time constant for component 1 for electrons | Time |
| ELECTRONSCINTILLATIONTIMECONSTANT2 | Constant | Time constant for component 2 for electrons | Time |
| ELECTRONSCINTILLATIONTIMECONSTANT3 | Constant | Time constant for component 3 for electrons | Time |

### Absorption

The implementation of optical photon bulk absorption, `G4OpAbsorption`, is trivial in that the process merely kills the particle. The procedure requires the user to fill the relevant `G4MaterialPropertiesTable` with empirical data for the absorption length, using `ABSLENGTH` as the property key. The absorption length is the average distance traveled by a photon before being absorbed by the medium; i.e., it is the mean free path returned by the `GetMeanFreePath` method.

Table 5.4: Material properties for the optical absorption process.

| Name | Type | Description | Unit category |
|---|---|---|---|
| ABSLENGTH | Energy-dependent | Absorption length | Length |

This `G4OpticalParameters` command can be used to configure the process.

- Set the verbosity of the absorption process. 0 = silent; 1 = initialisation; 2 = during tracking.
  - macro command: /process/optical/absorption/verbose 1
  - C++ statement: G4OpticalParameters::Instance()->SetAbsorptionVerboseLevel(G4int verboseLevel);
  - default value: 1

### Rayleigh Scattering

The differential cross section in Rayleigh scattering, $d\sigma/d\Omega$, is proportional to $1 + \cos^2\theta$, where $\theta$ is the angle of the new polarization vector with respect to the old polarization vector. The `G4OpRayleigh` scattering process samples this angle accordingly and then calculates the scattered photon's new direction by requiring that it be perpendicular to the photon's new polarization in such a way that the final direction, initial and final polarizations are all in one plane. This process thus depends on the particle's polarization (spin). A photon which does not have a polarization will not be Rayleigh scattered.

The process requires Rayleigh scattering attenuation length data. The Rayleigh scattering attenuation length is the average distance traveled by a photon before it is Rayleigh scattered in the medium and it is the distance returned by the `GetMeanFreePath` method. The attenuation length may be provided by the user filling a `G4MaterialPropertiesTable` with key `RAYLEIGH`.

The `G4OpRayleigh` class provides a method which can be used to calculate the attenuation coefficient of a medium following the Einstein-Smoluchowski formula. The derivation of this formula requires the use of statistical mechanics, includes temperature, and depends on the isothermal compressibility of the medium. This function is convenient when the Rayleigh attenuation length is not known from measurement but may be calculated from first principles using the above material constants. If the material property `RAYLEIGH` is not set, the attenuation coefficient will be calculated if either the material name is "Water", or the material's isothermal compressibility is provided using the material property `ISOTHERMAL_COMPRESSIBILITY`. For the material "Water", a temperature of 10ºC and isothermal compressibility of 7.658 x $10^{-23}$ $m^3$/MeV is used. For other materials, the temperature is determined by calling them material's `GetTemperature()`. The calculated attenuation length may be scaled by provided a material property `RS_SCALE_FACTOR`.

Table 5.5: Material properties for the optical Rayleigh scattering process.

| Name | Type | Description | Unit category |
|---|---|---|---|
| ISOTHERMAL_COMPRESSIBILITY | Constant | Isothermal compressibility. Can be used to calculate mean free path | Volume/Energy |
| RAYLEIGH | Energy-dependent | Attenuation length | Length |
| RS_SCALE_FACTOR | Constant | If set, multiply the calculated mean free path by this factor | Unitless |

This `G4OpticalParameters` command can be used to configure the process.

- Set the verbosity of the Rayleigh scattering process. 0 = silent; 1 = initialisation; 2 = during tracking.
  - macro command: /process/optical/rayleigh/verbose 1
  - C++ statement: G4OpticalParameters::Instance()->SetRayleighVerboseLevel(G4int verboseLevel);
  - default value: 1

### Wavelength Shifting

Wavelength shifting (WLS) fibers are used in many high-energy particle physics experiments. They absorb light at one wavelength and re-emit light at a different wavelength and are used for several reasons. For one, they tend to decrease the self-absorption of the detector so that as much light reaches the PMTs as possible. WLS fibers are also used to match the emission spectrum of the detector with the input spectrum of the PMT.

A WLS material is characterized by its photon absorption and photon emission spectrum and by a possible time delay between the absorption and re-emission of the photon. Wavelength Shifting may be simulated by specifying these empirical parameters for each WLS material in the simulation. It is sufficient to specify a relative spectral distribution as a function of photon energy for the WLS material. `WLSABSLENGTH` is the absorption length of the material as a function of the photon's energy. `WLSCOMPONENT` is the relative emission spectrum of the material as a function of the photon's energy, and `WLSTIMECONSTANT` accounts for any time delay which may occur between absorption and re-emission of the photon. An example is shown in the Listing 5.6.

Listing 5.6: Specification of WLS properties in `DetectorConstruction`.

```
std::vector<G4double> PhotonEnergy = { 6.6*eV, 6.7*eV, 6.8*eV, 6.9*eV,
                                  7.0*eV, 7.1*eV, 7.2*eV, 7.3*eV, 7.4*eV};

std::vector<G4double> RIndexFiber =
          {1.60, 1.60, 1.60, 1.60, 1.60, 1.60, 1.60, 1.60, 1.60};
std::vector<G4double> AbsFiber =
          {0.1*mm, 0.2*mm, 0.3*mm, 0.4*cm, 1.0*cm, 10.0*cm, 1.0*m, 10.0*m, 10.0*m};
std::vector<G4double> EmissionFiber =
          {0.0, 0.0, 0.0, 0.1, 0.5, 1.0, 5.0, 10.0, 10.0};

G4Material* WLSFiber = new G4Material(/*...*/);
G4MaterialPropertiesTable* MPTFiber = new G4MaterialPropertiesTable();

MPTFiber->AddProperty("RINDEX", PhotonEnergy, RIndexFiber);
MPTFiber->AddProperty("WLSABSLENGTH", PhotonEnergy, AbsFiber);
MPTFiber->AddProperty("WLSCOMPONENT", PhotonEnergy, EmissionFiber);
MPTFiber->AddConstProperty("WLSTIMECONSTANT", 0.5*ns);

WLSFiber->SetMaterialPropertiesTable(MPTFiber);
```

The way the `WLSTIMECONSTANT` is used depends on the time profile method chosen by the user. If the "exponential" option is set, the time delay between absorption and re-emission of the photon is sampled from an exponential distribution, with the decay term equal to WLSTIMECONSTANT. If, on the other hand, the "delta" option is chosen, the time delay is a delta function and equal to WLSTIMECONSTANT. The default is "delta".

The number of secondaries emitted may be configured. By default, each WLS interaction generates one secondary. If the material property WLSMEANNUMBERPHOTONS is set, the number of secondaries is chosen from a Poisson distribution with mean equal to WLSMEANNUMBERPHOTONS.

**Important:** New in GEANT4 version 10.7: definition of two WLS processes.

It is possible to have a material with two WLS processes. The G4OpticalPhysics constructor builds two processes by default, named OpWLS and OpWLS2. Usage of the second process is analogous to the first. The material property names for the OpWLS2 process are the same as for OpWLS, with a "2" appended. The time profile may be chosen to be "exponential" or "delta" using macro or C++ commands. In order to activate the OpWLS2 process in a material, the user needs to define the material properties.

Material properties are shown in Table 5.6.

Table 5.6: Material properties for WLS processes.

| Name | Type | Description | Unit category |
|---|---|---|---|
| WLSABSLENGTH | Energy-dependent | Absorption length of first WLS process | Length |
| WLSABSLENGTH2 | Energy-dependent | Absorption length of second WLS process | Length |
| WLSCOMPONENT | Energy-dependent | Emission spectrum of first WLS process | Unitless |
| WLSCOMPONENT2 | Energy-dependent | Emission spectrum of second WLS process | Unitless |
| WLSMEANNUMBER-PHOTONS | Constant | Mean number of photons emitted per interaction, for first WLS process | Unitless |
| WLSMEANNUMBER-PHOTONS2 | Constant | Mean number of photons emitted per interaction, for second WLS process | Unitless |
| WLSTIMECONSTANT | Constant | Time constant for emission, for first WLS process | Time |
| WLSTIMECONSTANT2 | Constant | Time constant for emission, for second WLS process | Time |

These parameters are available to configure the process:

- Set the time profile of the first WLS process to be either "delta" or "exponential":
  - macro command: /process/optical/wls/setTimeProfile value
  - C++ statement: G4OpticalParameters::Instance()->SetWLSTimeProfile(const G4String& val);
  - default value: delta
- Set the verbosity of the first WLS process. 0 = silent; 1 = initialisation; 2 = during tracking.
  - macro command: /process/optical/wls/verbose 1
  - C++ statement: G4OpticalParameters::Instance()->SetWLSVerboseLevel(G4int verboseLevel);
  - default value: 1
- Set the time profile of the second WLS process to be either "delta" or "exponential":
  - macro command: /process/optical/wls/setTimeProfile value
  - C++ statement: G4OpticalParameters::Instance()->SetWLSTimeProfile(const G4String& val);
  - default value: delta
- Set the verbosity of the second WLS process. 0 = silent; 1 = initialisation; 2 = during tracking.
  - macro command: /process/optical/wls/verbose 1
  - C++ statement: G4OpticalParameters::Instance()->SetWLSVerboseLevel(G4int verboseLevel);
  - default value: 1

### Mie Scattering

Mie Scattering (or Mie solution) is an analytical solution of Maxwell's equations for scattering of optical photons by spherical particles. It is significant only when the radius of the scattering object is of order of the wave length. The analytical expressions for Mie Scattering are complicated since they are a series sum of Bessel functions. One common approximation made is called *Henyey-Greenstein (HG)*. The implementation in GEANT4 follows the HG approximation (for details see the Physics Reference Manual) and the treatment of polarization and momentum are similar to that of Rayleigh scattering. We require the final polarization direction to be perpendicular to the momentum direction. We also require the final momentum, initial polarization, and final polarization to be in the same plane.

The process requires a `G4MaterialPropertiesTable` to be filled by the user with Mie scattering length data (entered with the name `MIEHG`) analogous to Rayleigh scattering. The Mie scattering attenuation length is the average distance traveled by a photon before it is Mie scattered in the medium and it is the distance returned by the GetMeanFreePath method. In practice, the user not only needs to provide the attenuation length of Mie scattering, but also needs to provide the constant parameters of the approximation $g_f$, $g_b$, and $r_f$, with `AddConstProperty` and with the names `MIEHG_FORWARD`, `MIEHG_BACKWARD`, and `MIEHG_FORWARD_RATIO`, respectively; see extended example optical/OpNovice.

Table 5.7: Material properties for the optical Mie scattering process.

| Name | Type | Description | Unit category |
|---|---|---|---|
| MIEHG | Energy-dependent | Attenuation length | Length |
| MIEHG_BACKWARD | Constant | Parameter used in sampling of scattering direction | Unitless |
| MIEHG_FORWARD | Constant | Parameter used in sampling of scattering direction | Unitless |
| MIEHG_FORWARD_RATIO | Constant | Parameter used in sampling of scattering direction | Unitless |

This `G4OpticalParameters` command can be used to configure the process.

- Set the verbosity of the Mie scattering process. 0 = silent; 1 = initialisation; 2 = during tracking.
  - macro command: /process/optical/mie/verbose 1
  - C++ statement: G4OpticalParameters::Instance()->SetMieVerboseLevel(G4int verboseLevel);
  - default value: 1

### Boundary Process

Optical photons interact with boundaries between volumes, for example, to reflect or refract. There are various methods of specifying a surface and assigning properties to it.

Reference: E. Hecht and A. Zajac, Optics [Hecht1974]

When a photon arrives at a medium boundary its behavior depends on the nature of the two materials that join at that boundary. Medium boundaries may be formed between two dielectric materials or a dielectric and a metal.

In the case of an interface between a dielectric and a metal, the photon can be absorbed by the metal or reflected back into the dielectric. If the photon is absorbed it can be detected according to the photoelectron efficiency of the metal.

As expressed in Maxwell's equations, Fresnel reflection and refraction are intertwined through their relative probabilities of occurrence. Therefore neither of these processes, nor total internal reflection, are viewed as individual processes deserving separate class implementation. Nonetheless, an attempt was made to adhere to the abstraction of having independent processes by splitting the code into different methods where practicable.

The program defaults to the GLISUR model and *polished* surface finish when no specific model and surface finish is specified by the user. In the case of a dielectric-metal interface, or when the GLISUR model is specified, the only surface finish options available are *polished* or *ground*. For dielectric-metal surfaces, the `G4OpBoundaryProcess` also defaults to unit reflectivity and zero detection efficiency. In cases where the user specifies the UNIFIED model (Fig. 5.1), but does not otherwise specify the model reflection probability constants, the default becomes Lambertian reflection.

When an optical photon arrives at a boundary it is absorbed if the medium of the volume being left behind has no index of refraction defined. A photon is also absorbed in case of a dielectric-dielectric polished or ground surface when the medium about to be entered has no index of refraction. It is absorbed for backpainted surfaces when the surface has no index of refraction.

The boundary process may produce warnings that can likely be ignored. These warnings result from the way the boundary process handles changes to which volume the particle is in. If the step length is 0, the boundary status is set to `StepTooSmall` and the appropriate group velocity is set. In practice, rather than comparing the step length to 0, the step is compared to the geometrical tolerance. For steps with length slightly over the geometric tolerance, it is not clear in the code logic which volume the photon is entering, so the group velocity is not set and a warning issued.

```
-------- WWWW ------- G4Exception-START -------- WWWW -------
*** G4Exception : OpBoun06
      issued by : G4OpBoundaryProcess
G4OpBoundaryProcess: Opticalphoton step length: 2.46916e-09 mm.
This is larger than the threshold 1e-09 mm to set status StepTooSmall.
Boundary scattering may be incorrect.

*** This is just a warning message. ***
-------- WWWW -------- G4Exception-END --------- WWWW -------
```

In most cases this warning may be ignored. It may be suppressed by setting optical verbosity, or boundary process verbosity, to 0.

### Specifying the surface

The physical surface object specifies which model the boundary process should use to simulate interactions with that surface. In addition, the physical surface can have a material property table all its own. The usage of this table allows all specular constants to be wavelength dependent. In case the surface is painted or wrapped (but not a cladding), the table may include the thin layer's index of refraction. This allows the simulation of boundary effects at the intersection between the medium and the surface layer, as well as the Lambertian reflection at the far side of the thin layer. This occurs within the process itself and does not invoke the `G4Navigator`. Combinations of surface finish properties, such as *polished* or *ground* and *front painted* or *back painted*, enumerate the different situations which can be simulated.

There are three methods of specifying an optical surface.

1. For the simple case of a perfectly smooth interface between two dielectric materials, all the user needs to provide are the refractive indices (`RINDEX`) of the two materials stored in their respective `G4MaterialPropertiesTable`s.
2. A *skin surface* is the surface entirely surrounding a logical volume. This is useful where a volume is coated with a reflector and placed into many different mother volumes. A limitation is that the skin surface can only have one and the same optical property for all of the enclosed volume's sides. An optical surface is created and defined using `G4OpticalSurface`. This surface is assigned to a logical volume using the class `G4LogicalSkinSurface`.
3. A *border surface* is defined by specifying the ordered pair of physical volumes touching at the surface. Because the pair of physical volumes is ordered, the user may specify different optical properties for photons arriving from the reverse side of the same interface. For the optical boundary process to use a border surface, the two volumes must have been positioned with `G4PVPlacement`. The ordered combination can exist at many

places in the simulation. An optical surface is created and defined using `G4OpticalSurface`. This surface is assigned to the ordered pair of physical volumes using the class `G4LogicalBorderSurface`.

If the geometry boundary has a *border surface* this surface takes precedence, otherwise the program checks for *skin surfaces*. The *skin surface* of the daughter volume is taken if a daughter volume is entered, or else the program checks for a *skin surface* of the current volume. When the optical photon leaves a volume without entering a daughter volume the *skin surface* of the current volume takes precedence over that of the volume about to be entered.

Listing 5.7: Defining border and skin surfaces (from extended example OpNovice).

```
#include "G4LogicalBorderSurface.hh"
#include "G4LogicalSkinSurface.hh"
#include "G4OpticalSurface.hh"

// The experimental Hall
//
G4Box* expHall_box = new G4Box("World", fExpHall_x, fExpHall_y, fExpHall_z);

G4LogicalVolume* expHall_log =
  new G4LogicalVolume(expHall_box, air, "World", 0, 0, 0);

G4VPhysicalVolume* expHall_phys =
  new G4PVPlacement(0, G4ThreeVector(), expHall_log, "World", 0, false, 0);

// The Water Tank
//
G4Box* waterTank_box = new G4Box("Tank", fTank_x, fTank_y, fTank_z);

G4LogicalVolume* waterTank_log =
  new G4LogicalVolume(waterTank_box, water, "Tank", 0, 0, 0);

G4VPhysicalVolume* waterTank_phys = new G4PVPlacement(
  0, G4ThreeVector(), waterTank_log, "Tank", expHall_log, false, 0);

// The Air Bubble
//
G4Box* bubbleAir_box = new G4Box("Bubble", fBubble_x, fBubble_y, fBubble_z);

G4LogicalVolume* bubbleAir_log =
  new G4LogicalVolume(bubbleAir_box, air, "Bubble", 0, 0, 0);

new G4PVPlacement(0, G4ThreeVector(0., 2.5 * m, 0.), bubbleAir_log, "Bubble",
                  waterTank_log, false, 0);

G4OpticalSurface* opWaterSurface = new G4OpticalSurface("WaterSurface");
opWaterSurface->SetType(dielectric_LUTDAVIS);
opWaterSurface->SetFinish(Rough_LUT);
opWaterSurface->SetModel(DAVIS);

G4LogicalBorderSurface* waterSurface = new G4LogicalBorderSurface(
  "WaterSurface", waterTank_phys, expHall_phys, opWaterSurface);

G4OpticalSurface* opAirSurface = new G4OpticalSurface("AirSurface");
opAirSurface->SetType(dielectric_dielectric);
opAirSurface->SetFinish(polished);
opAirSurface->SetModel(glisur);

G4LogicalSkinSurface* airSurface =
  new G4LogicalSkinSurface("AirSurface", bubbleAir_log, opAirSurface);
```

### Surface models: Defining the boundary properties

There are several models to describe the interactions of optical photons at a surface. These are defined in an emum:

Listing 5.8: Optical surface models.

```
enum G4OpticalSurfaceModel
{
  glisur,   // original GEANT3 model
  unified,  // UNIFIED model
  LUT,      // Look-Up-Table model (LBNL model)
  DAVIS,    // DAVIS model
  dichroic  // dichroic filter
};
```

The model is specified by calling the `SetModel(G4int model)` method of the surface.

### No surface defined

In the special case where no surface has been defined, but the two volumes defining the surface have the refractive index defined using the material property `RINDEX`, the surface is taken to be perfectly smooth, and both materials are taken to be dielectric. The photon can undergo total internal reflection, refraction or reflection, depending on the photon's wavelength, angle of incidence, and the refractive indices on both sides of the boundary. Furthermore, reflection and transmission probabilities are sensitive to the state of linear polarization.

### The UNIFIED model

One implementation of the `G4OpBoundaryProcess` class employs the UNIFIED model [Levin1996] of the DETECT program [Knoll1988]. It applies to dielectric-dielectric interfaces and tries to provide a realistic simulation, which deals with all aspects of surface finish and reflector coating. The surface may be assumed as smooth and covered with a metallized coating representing a specular reflector with given reflection coefficient, or painted with a diffuse reflecting material where Lambertian reflection occurs. The surfaces may or may not be in optical contact with another component and most importantly, one may consider a surface to be made up of micro-facets with normal vectors that follow given distributions around the nominal normal for the volume at the impact point. For very rough surfaces, it is possible for the photon to inversely aim at the same surface again after reflection of refraction and so multiple interactions with the boundary are possible within the process itself and without the need for relocation by `G4Navigator`.

The UNIFIED model (Fig. 5.1) provides for a range of different reflection mechanisms. The specular lobe constant (material property name `SPECULARLOBECONSTANT`) represents the reflection probability about the normal of a micro facet. The specular spike constant (material property name `SPECULARSPIKECONSTANT`), in turn, illustrates the probability of reflection about the average surface normal. The diffuse lobe constant is for the probability of internal Lambertian reflection, and finally the back-scatter spike constant (material property name `BACKSCATTERCONSTANT`) is for the case of several reflections within a deep groove with the ultimate result of exact back-scattering. The four probabilities add up to one, with the diffuse lobe constant being calculated by the code from other other three values that the user entered. The reader may consult the reference for a thorough description of the model.

It is possible to specify that a given fraction of photons are absorbed at the surface, or transmitted without change in direction or polarization. This is applicable for dielectric_dielectric interfaces that are not backpainted. The material properties `REFLECTIVITY` and `TRANSMITTANCE` are used. By default, `REFLECTIVITY` equals 1 and `TRANSMITTANCE` equals 0. At a surface interaction, a random number is chosen. If the random number is greater than the sum of the values of `REFLECTIVITY` and `TRANSMITTANCE` at the photon energy, the photon is absorbed. Otherwise, if the random number is greater than the `REFLECTIVITY` value, the photon is transmitted. Otherwise, the usual calculation of scattering takes place.

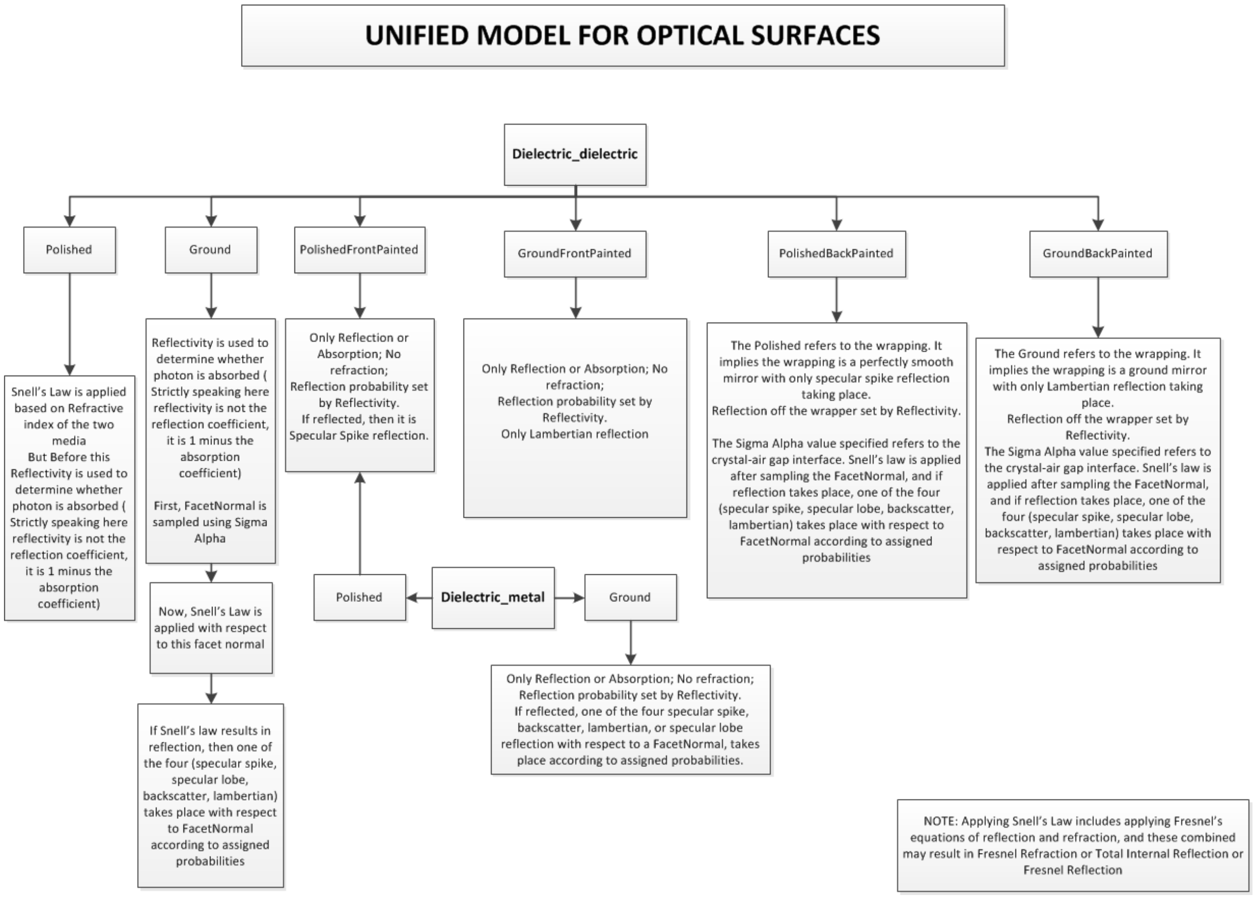

UNIFIED MODEL FOR OPTICAL SURFACES
Dielectric_dielectric
Polished
Ground
PolishedFrontPainted
GroundFrontPainted
PolishedBackPainted
GroundBackPainted
Snell's Law is applied based on Refractive index of the two media But Before this Reflectivity is used to determine whether photon is absorbed ( Strictly speaking here reflectivity is not the reflection coefficient, it is 1 minus the absorption coefficient)
Reflectivity is used to determine whether photon is absorbed ( Strictly speaking here reflectivity is not the reflection coefficient, it is 1 minus the absorption coefficient)
First, FacetNormal is sampled using Sigma Alpha
Now, Snell's Law is applied with respect to this facet normal
If Snell's law results in reflection, then one of the four (specular spike, specular lobe, backscatter, lambertian) takes place with respect to FacetNormal according to assigned probabilities
Only Reflection or Absorption; No refraction; Reflection probability set by Reflectivity. If reflected, then it is Specular Spike reflection.
Only Reflection or Absorption; No refraction; Reflection probability set by Reflectivity. Only Lambertian reflection
The Polished refers to the wrapping. It implies the wrapping is a perfectly smooth mirror with only specular spike reflection taking place. Reflection off the wrapper set by Reflectivity.
The Sigma Alpha value specified refers to the crystal-air gap interface. Snell's law is applied after sampling the FacetNormal, and if reflection takes place, one of the four (specular spike, specular lobe, backscatter, lambertian) takes place with respect to FacetNormal according to assigned probabilities
The Ground refers to the wrapping. It implies the wrapping is a ground mirror with only Lambertian reflection taking place. Reflection off the wrapper set by Reflectivity.
The Sigma Alpha value specified refers to the crystal-air gap interface. Snell's law is applied after sampling the FacetNormal, and if reflection takes place, one of the four (specular spike, specular lobe, backscatter, lambertian) takes place with respect to FacetNormal according to assigned probabilities
Polished
Dielectric_metal
Ground
Only Reflection or Absorption; No refraction; Reflection probability set by Reflectivity. If reflected, one of the four specular spike, backscatter, lambertian, or specular lobe reflection with respect to a FacetNormal, takes place according to assigned probabilities.
NOTE: Applying Snell's Law includes applying Fresnel's equations of reflection and refraction, and these combined may result in Fresnel Refraction or Total Internal Reflection or Fresnel Reflection

Fig. 5.1: Diagram of the UNIFIED Model for Optical Surfaces (courtesy A. Shankar)

If a photon is absorbed at the boundary, it may by detected–that is, its status set to “Detect” and its energy deposited locally. The material property `EFFICIENCY` is used to specify the fraction of photons detected.

The `sigma_alpha` parameter allows sepcification of the surface roughness. The facet normal is chosen from a Gaussian distribution with this sigma, with a maximum of the lower of 1 and 4 times sigma_alpha.

Listing 5.9: Dielectric-dielectric surface properties defined via the `G4OpticalSurface`.

```
G4VPhysicalVolume* volume1;
G4VPhysicalVolume* volume2;

G4OpticalSurface* OpSurface = new G4OpticalSurface("name");

G4LogicalBorderSurface* Surface = new
  G4LogicalBorderSurface("name",volume1,volume2,OpSurface);

OpSurface->SetType(dielectric_dielectric);
OpSurface->SetModel(unified);
OpSurface->SetFinish(groundbackpainted);
OpSurface->SetSigmaAlpha(0.1);

std::vector<G4double> pp = {2.038*eV, 4.144*eV};
std::vector<G4double> specularlobe = {0.3, 0.3};
std::vector<G4double> specularspike = {0.2, 0.2};
std::vector<G4double> backscatter = {0.1, 0.1};
std::vector<G4double> rindex = {1.35, 1.40};
std::vector<G4double> reflectivity = {0.3, 0.5};
std::vector<G4double> efficiency = {0.8, 0.1};

G4MaterialPropertiesTable* SMPT = new G4MaterialPropertiesTable();

SMPT->AddProperty("RINDEX", pp, rindex);
SMPT->AddProperty("SPECULARLOBECONSTANT", pp, specularlobe);
SMPT->AddProperty("SPECULARSPIKECONSTANT", pp, specularspike);
SMPT->AddProperty("BACKSCATTERCONSTANT", pp, backscatter);
SMPT->AddProperty("REFLECTIVITY", pp, reflectivity);
SMPT->AddProperty("EFFICIENCY", pp, efficiency);

OpSurface->SetMaterialPropertiesTable(SMPT);
```

### The Glisur model

The original GEANT3.21 implementation of this process is also available via the GLISUR methods flag, as shown in Listing 5.10. Note that there is considerable overlap of the Glisur and UNIFIED models in the code. The surface roughness is specified with the `polish` parameter.

Listing 5.10: Dielectric metal surface properties defined via the `G4OpticalSurface`.

```
G4LogicalVolume* volume_log;

G4OpticalSurface* OpSurface = new G4OpticalSurface("name");

G4LogicalSkinSurface* Surface = new
  G4LogicalSkinSurface("name",volume_log,OpSurface);

OpSurface->SetType(dielectric_metal);
OpSurface->SetFinish(ground);
OpSurface->SetModel(glisur);
OpSurface->SetPolish(0.8);
```

(continues on next page)

(continued from previous page)

```
G4MaterialPropertiesTable* OpSurfaceProperty = new G4MaterialPropertiesTable();

OpSurfaceProperty->AddProperty("REFLECTIVITY", pp, reflectivity);
OpSurfaceProperty->AddProperty("EFFICIENCY", pp, efficiency);

OpSurface->SetMaterialPropertiesTable(OpSurfaceProperty);
```

### LBNL look-up tables (LUT)

Janecek and Moses [Janecek2010] built an instrument for measuring the angular reflectivity distribution inside of BGO crystals with common surface treatments and reflectors applied. These results have been incorporated into the GEANT4 code. A third class of reflection type besides dielectric_metal and dielectric_dielectric is added: dielectric_LUT. The distributions have been converted to 21 look-up-tables (LUT); so far for 1 scintillator material (BGO) x 3 surface treatments x 7 reflector materials. The modified code allows the user to specify the surface treatment (rough-cut, chemically etched, or mechanically polished), the attached reflector (Lumirror, Teflon, ESR film, Tyvek, or TiO2 paint), and the bonding type (air-coupled or glued). The glue used is MeltMount, and the ESR film used is VM2000. Each LUT consists of measured angular distributions with 4º by 5º resolution in theta and phi, respectively, for incidence angles from 0º to 90º, in 1º-steps. The code might in the future be updated by adding more LUTs, for instance, for other scintillating materials (such as LSO or NaI). To use these LUT the user has to download them from Geant4 Software Download and set an environment variable, `G4REALSURFACEDATA`, to the directory of `geant4/data/RealSurface2.2`.

The enumeration G4OpticalSurfaceFinish includes:

```
polishedlumirrorair,         // mechanically polished surface, with lumirror
polishedlumirrorglue,        // mechanically polished surface, with lumirror & meltmount
polishedteflonair,           // mechanically polished surface, with teflon
polishedtioair,              // mechanically polished surface, with tio paint
polishedtyvekair,            // mechanically polished surface, with tyvek
polishedvm2000air,           // mechanically polished surface, with esr film
polishedvm2000glue,          // mechanically polished surface, with esr film & meltmount
etchedlumirrorair,           // chemically etched surface, with lumirror
etchedlumirrorglue,          // chemically etched surface, with lumirror & meltmount
etchedteflonair,             // chemically etched surface, with teflon
etchedtioair,                // chemically etched surface, with tio paint
etchedtyvekair,              // chemically etched surface, with tyvek
etchedvm2000air,             // chemically etched surface, with esr film
etchedvm2000glue,            // chemically etched surface, with esr film & meltmount
groundlumirrorair,           // rough-cut surface, with lumirror
groundlumirrorglue,          // rough-cut surface, with lumirror & meltmount
groundteflonair,             // rough-cut surface, with teflon
groundtioair,                // rough-cut surface, with tio paint
groundtyvekair,              // rough-cut surface, with tyvek
groundvm2000air,             // rough-cut surface, with esr film
groundvm2000glue             // rough-cut surface, with esr film & meltmount
```

To use an LBNL look-up-table, all the user needs to specify for an `G4OpticalSurface` is: `SetType(dielectric_LUT)`, `SetModel(LUT)` and for example, `SetFinish(polishedtyvekair)`.

Note that the LBNL look-up tables were the first optical surface LUT implemented in GEANT4. Where the term "LUT" is used, it often refers to the LBNL LUT.

### Davis look-up tables (LUTDAVIS)

Another model for optical surface interactions is called the LUT Davis model [RoncaliCherry2013], [Stockhoff2017], [Roncali2017]. The model is based on measured surface data and allows the user to choose from a list of available surface finishes. Provided are a rough and a polished L(Y)SO surface that can be used without reflector, or in combination with a specular reflector (e.g. ESR) or a Lambertian reflector (e.g. Teflon). The specular reflector can be coupled to the crystal with air or optical grease. Teflon tape is wrapped around the crystal with 4 layers.

Table 5.8: Surface names of available LUTs.

| | Bare | Teflon | ESR + Air | ESR + Optical Grease |
|---|---|---|---|---|
| Rough | Rough_LUT | RoughTeflon_LUT | RoughESR_LUT | RoughESRGrease_LUT |
| Polished | Polished_LUT | PolishedTeflon_LUT | PolishedESR_LUT | PolishedESRGrease_LUT |

In the LUT database, typical roughness parameters obtained from the measurements are provided to characterize the type of surface modelled:

$$\begin{aligned}\text{ROUGH}: R_a = 0.48\mu\text{m}, \sigma = 0.57\mu\text{m}, R_{pv} = 3.12\mu\text{m}\\ \text{POLISHED}: R_a = 20.8\text{ nm}, \sigma = 26.2\text{ nm}, R_{pv} = 34.7\text{ nm}\end{aligned} \tag{5.1}$$

with $R_a$ = average roughness; $\sigma$ = rms roughness, $R_{pv}$ = peak-to-valley ratio.

The detector surface, called `Detector_LUT`, defines a polished surface coupled to a photodetector with optical grease or a glass interface (similar index of refraction 1.5). To use `Detector_LUT`, the surface property `EFFICIENCY` must be greater than 0. Any surface can be used as a detector surface when the `EFFICIENCY` is set to 1.

To enable the LUT Davis Model, the user needs to specify for a `G4OpticalSurface`: `SetType(dielectric_LUTDAVIS)`, `SetModel(DAVIS)` and also, for example, `SetFinish(Rough_LUT)`. The user also has to download data files from Geant4 Software Download and set an environment variable, `G4REALSURFACEDATA`, to the directory of `geant4/data/RealSurface2.2`.

*Background*

The crystal topography is obtained with atomic force microscopy (AFM). From the AFM data, the probability of reflection (1) and the reflection directions (2) are computationally determined, for incidence angles ranging from 0° to 90°. Each LUT is computed for a given surface and reflector configuration. The reflection probability in the LUT combines two cases: directly reflected photons from the crystal surface and photons that are transmitted to the reflector surface and later re-enter the crystal. The key operations of the reflection process are the following: The angle between the incident photon (Old Momentum) and the surface normal are calculated. The probability of reflection is extracted from the first LUT. A Bernoulli test determines whether the photon is reflected or transmitted. In case of reflection two angles are drawn from the reflection direction LUT.

### Thin film optical coatings

L.Cappellugola, M. Dupont, S. Curtoni and C. Morel, Aix Marseille Univ.

Photons can be transmitted through a thin film that has a thickness of the order of light wavelength. Interference phenomena and frustrated transmission beyond the limit angle have then to be considered.

New in version 11.1, Geant4can account for thin film interfaces. This is achieved by defining a new method that provides a transmission probability for optical tracking of optical photons. This new method is called `CoatedDielectricDielectric()` and considers only two physical and logical media, and the refractive index and thickness of the thin film separating these two media.

*Reflectance and transmittance of an optical coating*

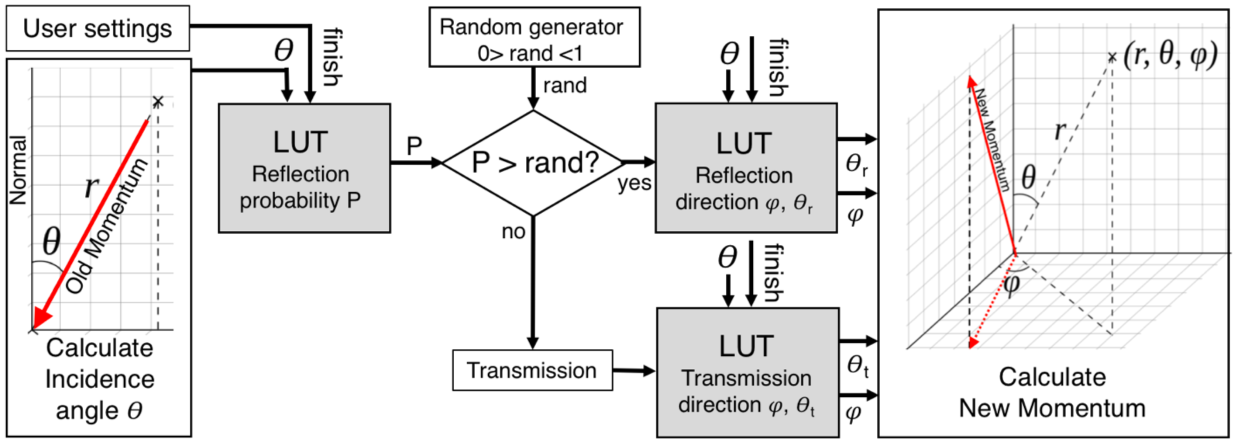


Fig. 5.2: Old Momentum to New Momentum. The old momentum is the unit vector that describes the incident photon. The reflected/transmitted photon is the New Momentum described by two angles $\phi$ and $\theta$.

Let $r_{ij}$ be the Fresnel coefficient describing an interface between media $i$ and $j$, with refractive indices $n_i$ and $n_j$. If the incidence angle $\theta_i$ is greater than the limit angle $\theta_{\text{lim}}$, the term $n_j cos(\theta_j)$ in the Fresnel coefficients with $n_i sin(\theta_i) = n_j sin(\theta_j)$ will be replaced by $i\gamma = \sqrt{n_i^2 \sin{\theta_i}^2 - n_j^2}$. Hence we get for $\theta > \theta_l$:

$$r_{TE} = \frac{n_i cos(\theta_i) - i\gamma}{n_i cos(\theta_i) + i\gamma}$$

$$r_{TM} = \frac{n_i i\gamma - n_j^2 cos(\theta_i)}{n_i i\gamma + n_j^2 cos(\theta_i)}$$

In case of a transmission from medium 1 to medium 3 through a thin passivation film (medium 2), the reflection coefficient for incidence angles lesser and greater than the limit angle $\theta_{\text{lim}} = \arcsin(n_2/n_1)$ is given by:

$$r_{\theta<\theta_{\text{lim}}} = \frac{r_{12} + r_{23}e^{2i\beta'}}{1 + r_{12}r_{23}e^{2i\beta'}} \tag{5.2}$$

$$r_{\theta>\theta_{\text{lim}}} = \frac{r_{12} + r_{23}e^{2\beta''}}{1 + r_{12}r_{23}e^{2\beta''}} \tag{5.3}$$

where $\beta' = k_2 d \cos\theta_2$ and $\beta'' = -kd\gamma$ with $d$ the layer thickness and $k_2 = 2\pi/\lambda_2 = 2\pi n_2/\lambda$ the photon wavenumber in medium 2 with $\lambda$ and $k$ the photon wavelength and wavenumber in vacuum.

*Integration of frustrated transmission in Geant4*

The main method of `G4OpBoundaryProcess` class is the `PostStepDoIt()` method. This method starts by getting refractive indices of both media on each side of the interface, instantiating the `Type`, `Model` and `Finish` properties of this optical surface and calling the corresponding function in order to analyse the interaction on the interface correctly.

A new surface type named `coated` is used for an interface between two dielectric volumes separated by a thin film and a new method named `CoatedDielectricDielectric()`, which is called by `PostStepDoIt()`. Only the two volumes on each side of the thin film have to be physically and logically defined. The surface can be parametrized with two arrays named `CoatedRindex` and `CoatedThickness`, which correspond to the thin film layer refractive index and thickness, respectively, as a function of wavelength. Frustrated transmission for incidence angles superior to the limit angle can be enabled or disabled via the boolean parameter `CoatedFrustratedTransmission` in order to assess the impact of frustrated transmission.

An example is given with the extended example OpNovice2 and macro coated.mac.

*Results of the implementation*

*Transmittance through a thin optical coating*

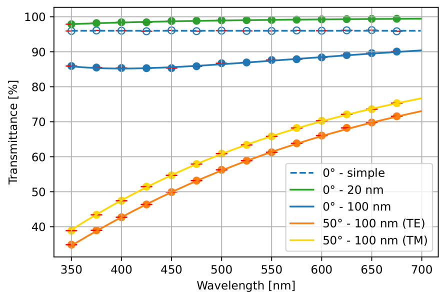


Fig. 5.3: Transmittance as a function of wavelength through an interface with a thin film. In blue and green: transmittances for a normal incidence, in orange and yellow: transmittances for a 50° incidence. Curves represent theoretical transmittances for a simple interface (dotted line) and for a thin film interface (continuous lines). Markers represent the transmittance estimated by the Geant4 simulation for a simple interface (empty dots) and for a thin film interface (full dots).

We fix $n_1 = n_3 = 1.5$ and $n_2 = 1.0$ (the limit angle between medium 1 and 2 is $\theta_l = 41.8°$. Two examples are presented in Fig. 5.3 for different coating thicknesses $d$ and incidence angle $\theta$. The first example stands for a simple interface, comprising only one dioptre from medium 1 to medium 2. The theoretical transmittance, which is given by:

$$T = 1 - R = 1 - rr^* \tag{5.4}$$

with $r = r_{TE} = r_{TM}$ given by the equations above for $\theta_i = \theta_j = 0$ corresponding to a normal incidence, is presented with dashed blue line and the fraction of photons transmitted through this dioptre estimated by the Geant4 simulation of 2,800,000 visible photons are represented by blue empty dots. The second example highlights the consequences of light transmission through a thin layer by adding a medium 3 after the medium 2 in such a way that the thickness d of the medium 2 is smaller than the wavelength of the light. Theoretical transmittances for different thicknesses $d$ and incidence angles $\theta$ ($\theta = 0$ for a normal incidence and $\theta = 50°$ for an incidence angle superior to the limit angle $\theta_{\text{lim}}$) are represented with continuous lines. The fraction of photons transmitted through these different interfaces estimated by the Geant4 simulation of 2,800,000 visible photons are represented by full dots.

In case of an oblique incidence, the transmittance for both the Transverse Electric (TE) and the Transverse Magnetic (TM) polarizations are shown and compared to the theoretical transmittances using (5.4) with $r = r_{TE}$ or $r = r_{TM}$, respectively. For an incidence angle superior to the limit angle (orange curves), the transmittance is non-zero because of the frustrated transmission of light.

The perfect agreement between the simulated and theoretical transmittances assess the correct implementation of the model in the Geant4 software.

*Transmittance through a window optically coated on both its faces*

In this section we model two successive interfaces at the input and output faces of a window of refractive index $n_3 = 2.0$ that are optically coated with 100 nm of an oxide of refractive index $n_2 = 1.5$. This coated window is placed in air ($n_1 = 1.0$).

For a normal incidence, the theoretical transmittance involving multiples reflections (MR) of light inside the window is estimated by $T_{MR}$:

$$T_{MR} = \frac{T^2}{1 - R^2}$$

where $R = rr^*$ is the reflectance with $r$ given by the Eqs. (5.2) or (5.3) and $T = 1 - R$.

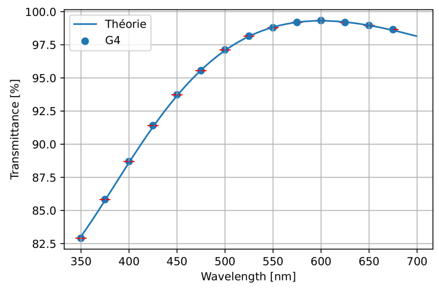


Fig. 5.4: Transmittance for normal incidence through a coated window ($n_1 = 1.0$, $n_2 = 1.5$ and $n_3 = 2.0$). The continuous line represents the theoretical transmittance and the markers represents the transmittance estimated by the Geant4 simulation of 2,800,000 visible photons.

Fig. 5.4 shows the transmittance for a normal incidence through the coated window estimated by the simulation of 2,800,000 visible photons. Here again, the perfect agreement between the simulated and theoretical transmittances assess the correct implementation of the model in the Geant4software.

### Complex index of refraction

The reflectivity off a metal surface can also be calculated by way of a complex index of refraction. Instead of storing the `REFLECTIVITY` directly, the user stores the real part (`REALRINDEX`) and the imaginary part (`IMAGINARYRINDEX`) as a function of photon energy separately in the G4MaterialPropertyTable. GEANT4 then calculates the reflectivity depending on the incident angle, photon energy, degree of TE and TM polarization, and this complex refractive index.

If it is desired that photons are not absorbed at the surface, but instead are transmitted into the material, the user can set the transmittance (TRANSMITTANCE) to 1. If this setting is used together with a ground finish and a non-zero value of sigma_alpha, some photons are still absorbed at the surface. If the absorption at the surface should be completely prevented, sigma_alpha must be zero or a polished finish needs to be used. Additionally, the absorption length (ABSLENGTH) should be defined for the material, otherwise the material would be fully transparent. Note that Geant4 does not calculate the absorption length, which is a material property, from the imaginary part of the refractive index, which is a surface property. The user needs to provide both properties individually. The macro complexRindex.mac (below) in the example OpNovice2 demonstrates the usage of the complex refractive index.

Listing 5.11: A macro for use with OpNovice2 demonstrating the usage of the complex index of refraction.

```
/control/verbose 2
/tracking/verbose 0
/run/verbose 0
/process/optical/verbose 1
/control/cout/ignoreThreadsExcept 0

/opnovice2/worldMaterial G4_Al
/opnovice2/worldProperty GROUPVEL  0.000002 299.792     0.000008 299.792
/opnovice2/worldProperty RINDEX    0.000002 1.38        0.000004 0.28        0.000006 0.13       ␣
↪ 0.000008 0.077
/opnovice2/worldProperty ABSLENGTH 0.000002 6.75e-9     0.000004 6.65e-9     0.000006 6.96e-9    ␣
↪ 0.000008 7.45e-9

/opnovice2/boxMaterial G4_AIR
/opnovice2/boxProperty RINDEX    0.000002 1.01    0.000008 1.01
/opnovice2/boxProperty ABSLENGTH 0.000002 1000000 0.000005 2000000 0.000008 3000000
```

(continues on next page)

(continued from previous page)

```
/opnovice2/surfaceModel unified
/opnovice2/surfaceType dielectric_metal
/opnovice2/surfaceFinish polished
/opnovice2/surfaceSigmaAlpha 0 # only relevant for ground finish

/opnovice2/surfaceProperty REALRINDEX           0.000002 1.38     0.000004 0.28     0.000006 0.
↪13     0.000008 0.077
/opnovice2/surfaceProperty IMAGINARYRINDEX      0.000002 7.31     0.000004 3.71     0.000006 2.
↪40     0.000008 1.71
/opnovice2/surfaceProperty TRANSMITTANCE        0.000002 1        0.000006 1
/opnovice2/surfaceProperty SPECULARLOBECONSTANT 0.000002 1        0.000006 1 # only relevant for␣
↪ground finish

/run/initialize

/gun/particle opticalphoton
/gun/energy 4 eV
/gun/position 90 0 0 cm
/gun/direction 1 0 0
/opnovice2/gun/optPhotonPolar 0. deg
/opnovice2/gun/randomDirection true

/analysis/setFileName complexRindex_0
/analysis/h1/set 20 90 0 90  # deg
/analysis/h1/set 21 90 0 90  # deg
/run/printProgress 100000
/run/beamOn 500000

/analysis/setFileName complexRindex_90
/opnovice2/gun/optPhotonPolar 90. deg
/run/beamOn 500000
```

### Dichroic filter

A dielectric-dichroic boundary may be specified by supplying a dichroic vector to the optical surface.

## Configuration

Table 5.9: Material and surface properties for the optical boundary scattering process.

| Name | Type | Material or surface | Description |
|---|---|---|---|
| BACKSCATTER-CONSTANT | Energy-dependent | Surface | Unified model |
| EFFICIENCY | Energy-dependent | Surface | Chance of an absorbed photon to be detected |
| COATEDFRUS-TRATEDTRANS-MISSION | Constant | Surface | Whether to use frustrated transmission for thin coatings. Values are either 1 (on) or 0 (off) |
| GROUPVEL | Energy-dependent | Material | If specified, set the photon velocity to this value on entering the material, if undergoing Fresnel refraction or transmission |
| IMAGI-NARYRINDEX | Energy-dependent | Either | Imaginary part of complex refractive index |
| REFLECTIVITY | Energy-dependent | Surface | 1 minus the absorption coefficient |
| REALRINDEX | Energy-dependent | Either | Real part of complex refractive index |
| RINDEX | Energy-dependent | Either | Refractive index |
| COATEDRINDEX | Energy-dependent | Surface | Refractive index of thin coating |
| SPECULARLOBE-CONSTANT | Energy-dependent | Surface | Unified model |
| SPECULARSPIKE-CONSTANT | Energy-dependent | Surface | Unified model |
| SURFACEROUGH-NESS | Constant | Surface | Parameter to express surface roughness for dielectric_dielectric surfaces, leading to Lambertian scattering |
| COATEDTHICK-NESS | Energy-dependent | Surface | Thickness of thin coating |
| TRANSMITTANCE | Energy-dependent | Surface | For dielectric_dielectric surfaces |

These `G4OpticalParameters` commands can be used to configure the process.

- Set the verbosity of the boundary scattering process. 0 = silent; 1 = initialisation; 2 = during tracking.
  - macro command: /process/optical/boundary/verbose 1
  - C++ statement: G4OpticalParameters::Instance()->SetBoundaryVerboseLevel(G4int verboseLevel);
  - default value: 1
- Call the sensitive detector automatically.
  - macro command: /process/optical/boundary/setInvokeSD
  - C++ statement: G4OpticalParameters::Instance()->SetBoundaryInvokeSD(G4bool val)
  - default value: false

### 5.2.8 Parameterisation

In this section we describe how to use the parameterisation or "fast simulation" facilities of GEANT4. Examples are provided in the **examples/extended/parameterisations** directory.

#### Generalities:

The GEANT4 parameterisation facilities allow you to shortcut the detailed simulation in a given volume and for given particle types in order for you to provide your own implementation of the physics and of the detector response. This allows to make an alternative modelling of the physics processes, usually approximate and faster than the detailed simulation.

Overtaking the detailed GEANT4 simulation (tracking) requires the user to specify 3 main components of the parameterisation:

1. Where the particles are parameterised;
2. Which particles are parameterised:
    - static conditions (particle type, PDG, charge, . . . )
    - dynamic conditions (energy, direction, . . . )
3. What happens instead of the detailed simulation:
    - where the particle is moved
    - what are the created secondaries
    - is the primary particle killed
    - what (and where) energy is deposited

    *Note: It is user responsibility to invoke Hit method of the sensitive detector*

#### 1. Where

Parameterisations are bound to a `G4Region` object, which, in the case of fast simulation can be called an **envelope**. A `G4Region` has a `G4LogicalVolume` object (or a series of `G4LogicalVolume` objects) as a root, and the `G4Region` is attached to this volume and all its ancestors:

```
G4Region(const G4String&) // constructor also registers to G4RegionStore
void G4Region::AddRootLogicalVolume (G4LogicalVolume*) // attach root volume to region
```

Envelopes often correspond to the outer volumes of (sub-)detectors: electromagnetic calorimeters, tracking chambers, etc.

With GEANT4 it is also possible to define envelopes by overlaying a parallel ("ghost") geometry as discussed in *Parameterisation Using Ghost/Parallel Geometries*.

#### 2. Which particles

Parameterisation is usually specified only for certain particles. Those particles must have attached `G4FastSimulationManagerProcess` to their list of processes. For users of modular physics lists (`G4VModularPhysicsList`) — from which reference physics lists (`FTFP_BERT`, `QGSP_BIC`,. . . ) are derived — it is enough to use the helper class `G4FastSimulationPhysics` and activate the parameterisation for each particle type:

```
FTFP_BERT* physicsList = new FTFP_BERT; // G4VModularPhysicsList
auto fastSimulationPhysics = new G4FastSimulationPhysics(); // helper
fastSimulationPhysics->ActivateFastSimulation("e-"); // for envelope in mass geometry
```

(continues on next page)

(continued from previous page)

```
fastSimulationPhysics->ActivateFastSimulation("pi+","pionGhostWorld"); // for envelopes in␣
↪parallel geometry
physicsList->RegisterPhysics( fastSimulationPhysics ); // attach to the physics list
```

GEANT4 will take into account the parameterisation process at tracking for any step that starts in any root `G4LogicalVolume` of the `G4Region` that has been declared as an envelope. It will proceed by first asking the available parameterisation models for that particle (models with static conditions fulfilled by that particle). Existing models will be checked (in the order they were added) if one of them (and only one) wants to issue a trigger (meaning the dynamic conditions are met). If yes, it will invoke its parameterisation. In this case, the tracking and physics (detailed simulation) **will not apply be applied** to the particle in the step. In case no trigger is issued, the simulation continues as usual, with other processes being taken into account.

Parameterisations resembles a "user stepping action" but are more advanced because:

- parameterisation code is considered only in the `G4Region` to which it is bound;
- parameterisation code is considered anywhere in the `G4Region`, that is, any volume in which the track is located (mother, daughter, sub-daughter, . . . );
- GEANT4 will provide information to your parameterisation code about the current root volume of the `G4Region` in which the track is travelling.

### 3. What happens

Implementation of the parameterisation must be made deriving from the `G4VFastSimulationModel` abstract class. Models are attached to the `G4Region` and will be considered only if particle meets the selection criteria and is within the geometrical hierarchy tree of the root logical volume of the `G4Region`. The `G4Region` is specified in the model constructor, together with the model name:

```
// constructor adds this model to G4FastSimulationManager of given envelope
G4VFastSimulationModel(const G4String&, G4Region*)
```

Selection criteria are to be defined in `G4VFastSimulationModel` implementation:

```
// specify the static conditions (particle type, PDG, charge, ...)
virtual G4bool G4VFastSimulationModel::IsApplicable (const G4ParticleDefinition&) = 0
// specify the dynamic conditions (momentum, direction, position, distance to boundary, ...)
virtual G4bool G4VFastSimulationModel::ModelTrigger (const G4FastTrack&) = 0
```

As particle is not transported by GEANT4, it is therefore responsibility of the model to describe what happens to the particle in place of the standard simulation: where the particle goes to (or to kill it), how its momentum is changed, what is the deposited energy (user needs to register manually all deposits in the sensitive detector or use helper class `G4FastSimHitMaker`), and what secondary particles are created. Those details should be implemented within:

```
// input information: G4FastTrack
// output information: G4FastStep
virtual G4bool G4VFastSimulationModel::DoIt(const G4FastTrack&, G4FastStep&) = 0
```

GEANT4 contains an implementation of the Gflash parameterisation model (see more in *Gflash Parameterisation*) and several examples in `extended/parameterisations` directory.

### Overview of Parameterisation Components

The GEANT4 components which allow the implementation and control of parameterisations are:

**G4VFastSimulationModel** This is the abstract class for the implementation of parameterisations. You must inherit from it to implement your concrete parameterisation model.

**G4Region** As mentioned before, an envelope of parameterisation in GEANT4 is a `G4Region`. The parameterisation is bound to the `G4Region` by setting a `G4FastSimulationManager` pointer to it.

Fig. 5.5 shows how the `G4VFastSimulationModel` and `G4FastSimulationManager` objects are bound to the `G4Region`. Then for all root G4LogicalVolume's held by the G4Region, the fast simulation code is active.

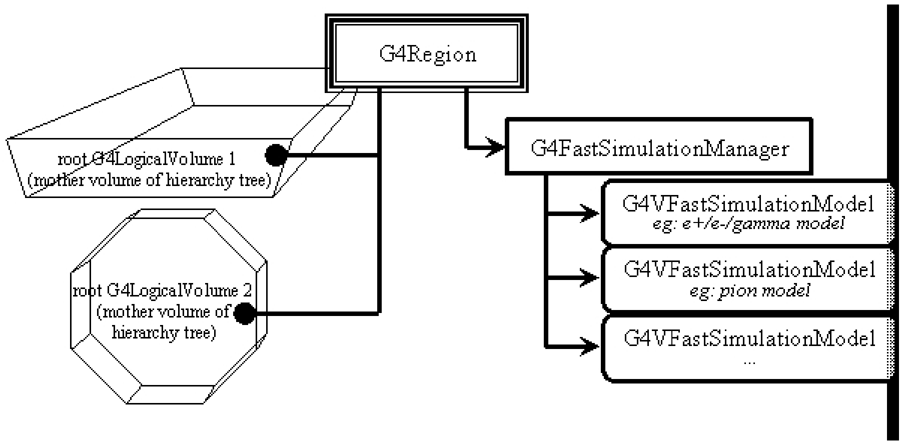


Fig. 5.5: `G4VFastSimulationModel` and `G4FastSimulationManager` objects.

**G4FastSimulationManager** The `G4VFastSimulationModel` objects are attached to the `G4Region` through a `G4FastSimulationManager`. This object will manage the list of models and will message them at tracking time.

**G4FastSimulationManagerProcess** This is an implementation of `G4VProcess`. It invokes the parameterisation models if trigger conditions are met (particle is within an envelope, of certain type, etc.). It must be set in the process list of the particles you want to parameterise (e.g. using physics constructor `G4FastSimulationPhysics` on top of any modular physics list - since 10.3 release). If added manually, one must remember that in presence of the parallel world, the ordering of processes matters.

**G4GlobalFastSimulationManager** This a singleton class which provides the management of the `G4FastSimulationManager` objects and some ghost facilities.

**G4FastSimHitMaker** This is a helper class that can be employed in the fast simulation models. It allows to deposit energy at given position (`G4FastHit`), provided it is located within the sensitive detector that derives from `G4VFastSimSensitiveDetector` base class. An extended example **extended/parameterisations/Par03** demonstrates how to use `G4FastSimHitMaker` to create multiple deposits from the fast simulation model. Such model should create energy deposits, and for each one call the method `G4FastSimHitMaker::make(const G4FastHit& aHit, const G4FastTrack& aTrack)` so that sensitive detector can be located, and hits stored in hit collection, as implemented in the sensitive detector class user implementation.

**G4VFastSimSensitiveDetector** This is a base class for a sensitive detector that allows to easily store hits created in the fast simulation models. It must me used in addition to inheritance from the usual base class *G4VSensitiveDetector* for processing of energy deposited in G4Step. ProcessHits(…) method must be implemented and describe how hits should be saved in the hit collections. It is invoked by Hit method which is public and can be called directly in the fast simulation model (if sensitive detector is known to the model), or via the helper class `G4FastSimHitMaker` that will locate appropriate volume and retrieve its sensitive detector. An extended example **extended/parameterisations/Par03** demonstrates how to use `G4VFastSimSensitiveDetector` to deposit energy from fast simulation and compare it to the detailed simulation.

### The `G4VFastSimulationModel` Abstract Class

**Constructors**

The `G4VFastSimulationModel` class has two constructors.

**`G4VFastSimulationModel(const G4String& aName):`** Here `aName` identifies the parameterisation model.

**`G4VFastSimulationModel(const G4String& aName, G4Region*, G4bool IsUnique=false):`** In addition to the model name, this constructor accepts a `G4Region` pointer. The needed `G4FastSimulationManager` object is constructed if necessary, passing to it the G4Region pointer and the Boolean value. If it already exists, the model is simply added to this manager. Note that the `G4VFastSimulationModel` object *will not keep track* of the `G4Region` passed in the constructor. The Boolean argument is there for optimisation purposes: if you know that the `G4Region` has a unique root `G4LogicalVolume`, uniquely placed, you can set the Boolean value to `true`.

**Virtual methods**

The `G4VFastSimulationModel` has pure virtual methods which must be overridden in your concrete class:

**`G4VFastSimulationModel(const G4String& aName):`** Here aName identifies the parameterisation model.

**`G4bool ModelTrigger( const G4FastTrack&):`** You must return `true` when the dynamic conditions to trigger your parameterisation are fulfilled. `G4FastTrack` provides access to the current `G4Track`, gives simple access to the current root `G4LogicalVolume` related features (its `G4VSolid`, and `G4AffineTransform` references between the global and the root `G4LogicalVolume` local coordinates systems) and simple access to the position and momentum expressed in the root `G4LogicalVolume` coordinate system. Using these quantities and the `G4VSolid` methods, you can for example easily check how far you are from the root `G4LogicalVolume` boundary, or if the particle is entering or escaping the volume.

**`G4bool IsApplicable(const G4ParticleDefinition&):`** In your implementation, you must return `true` when your model is applicable to the `G4ParticleDefinition` passed to this method. The `G4ParticleDefinition` provides all intrinsic particle information (mass, charge, spin, name . . . ).
If you want to implement a model which is valid only for certain particle types, it is recommended for efficiency that you use the static pointer of the corresponding particle classes.
As an example, in a model valid for *gamma*s only, the `IsApplicable()` method should take the form:

```
#include "G4Gamma.hh"

G4bool MyGammaModel::IsApplicable(const G4ParticleDefinition& partDef)
{
  return &partDef == G4Gamma::GammaDefinition();
}
```

**`void DoIt(const G4FastTrack&, G4FastStep&):`** The details of your parameterisation will be implemented in this method. The `G4FastTrack` reference provides the input information, and the final state of the particles after parameterisation must be returned through the `G4FastStep` reference (what is the changed energy and position, what are the secondaries, which particles are killed). Tracking for the final state secondary particles is requested after your parameterisation has been invoked.

### The `G4FastSimulationManager` Class

`G4FastSimulationManager` functionalities regarding the use of ghost volumes are explained in *Parameterisation Using Ghost/Parallel Geometries*.

**Constructor**

**`G4FastSimulationManager(G4Region *anEnvelope, G4bool IsUnique=false):`** This is the only constructor. You specify the `G4Region` by providing its pointer. The `G4FastSimulationManager` object will bind itself to this `G4Region`. If you know that this `G4Region` has a single root `G4LogicalVolume`, placed only once, you can set the `IsUnique` boolean to `true` to allow some optimisation.

Note that if you choose to use the `G4VFastSimulationModel(const G4String&, G4Region*, G4bool)` constructor for your model, the `G4FastSimulationManager` will be constructed using the given `G4Region*` and `G4bool` values of the model constructor.

**Management of parameterisation models**

The following two methods provide the usual management functions.

- `void AddFastSimulationModel( G4VFastSimulationModel*)`
- `void RemoveFastSimulationModel( G4VFastSimulationModel*)`

**Messenger**

To ease the communication with `G4FastSimulationManager` a messenger class was introduced. List of available commands:

```
/param/ // Fast Simulation print/control commands.
/param/showSetup // Show fast simulation setup (for each world: fast simulation manager
                 // process - which particles, region hierarchy - which models)
/param/listEnvelopes <ParticleName (default:all)> // List all the envelope names for a
                         // given particle (or for all particles if without parameters).
/param/listModels <EnvelopeName (default:all)> // List all the Model names for a given
                                 // envelope (or for all envelopes if without parameters).
/param/listIsApplicable <ModelName (default:all)> // List all the Particle names a given
                        // model is applicable (or for all models if without parameters).
/param/ActivateModel <ModelName> // Activate a given Model.
/param/InActivateModel <ModelName> // InActivate a given Model.
```

### The `G4FastSimulationManagerProcess` Class

This `G4VProcess` serves as an interface between the tracking and the parameterisation. You usually don't need to set it up directly, as you can conveniently rely on the helper tool `G4FastSimulationPhysics` (see Section *2. Which particles*) for this. At tracking (stepping) time, it collaborates with the `G4FastSimulationManager` of the current volume, if any, to allow the models to trigger. If no manager exists or if no model issues a trigger, the tracking goes on normally.

**Parallel/Ghost Geometry**

In order to register `G4FastSimulationManagerProcess` operating on the regions from the parallel geometry, it must be constructed passing the world name. Otherwise, the mass geometry is used (and its navigator).

Moreover, `G4FastSimulationManagerProcess` must be registered to `G4ProcessManager` as continuous and discrete process, as it provides navigation in the ghost world to limit the step on ghost boundaries. Hence it is important to maintain the ordering of the along-step execution:

```
[n-3] ...
[n-2] Multiple Scattering
[n-1] G4FastSimulationManagerProcess
[ n ] G4Transportation
```

Since 10.3 release it is convenient to use `G4FastSimulationPhysics` on top of a modular physics list. To use fast simulation in the parallel geometry (register `G4FastSimulationManagerProcess`) one can do (from Par01 example):

```
// -----------------------------------------------
// -- PhysicsList and fast simulation activation:
// -----------------------------------------------
// -- Create a physics list (note : FTFP_BERT is a G4VModularPhysicsList
// -- which allows to use the subsequent G4FastSimulationPhysics tool to
// -- activate the fast simulation):
FTFP_BERT* physicsList = new FTFP_BERT;
// -- Create helper tool, used to activate the fast simulation:
G4FastSimulationPhysics* fastSimulationPhysics = new G4FastSimulationPhysics();
fastSimulationPhysics->BeVerbose();
// -- activation of fast simulation for particles having fast simulation models
// -- attached in the mass geometry:
fastSimulationPhysics->ActivateFastSimulation("e-");
fastSimulationPhysics->ActivateFastSimulation("e+");
fastSimulationPhysics->ActivateFastSimulation("gamma");
// -- activation of fast simulation for particles having fast simulation models
// -- attached in the parallel geometry:
fastSimulationPhysics->ActivateFastSimulation("pi+","pionGhostWorld");
fastSimulationPhysics->ActivateFastSimulation("pi-","pionGhostWorld");
// -- Attach the fast simulation physics constructor to the physics list:
physicsList->RegisterPhysics( fastSimulationPhysics );
// -- Finally passes the physics list to the run manager:
runManager->SetUserInitialization(physicsList);
```

If you wish to register manually `G4FastSimulationManagerProcess` to all the particles as a discrete and continuous process:

```
void MyPhysicsList::addParameterisation()
{
  G4FastSimulationManagerProcess*
    theFastSimulationManagerProcess = new G4FastSimulationManagerProcess();
  theParticleIterator->reset();
  while( (*theParticleIterator)() )
    {
      G4ParticleDefinition* particle = theParticleIterator->value();
      G4ProcessManager* pmanager = particle->GetProcessManager();
      pmanager->AddProcess(theFastSimulationManagerProcess, -1, 0, 0);
    }
}
```

### The `G4GlobalFastSimulationManager` Singleton Class

This class is a singleton which can be accessed as follows:

```
#include "G4GlobalFastSimulationManager.hh"
...
...
G4GlobalFastSimulationManager* globalFSM;
globalFSM = G4GlobalFastSimulationManager::getGlobalFastSimulationManager();
...
...
```

Presently, you will mainly need to use the `G4GlobalFastSimulationManager` if you use ghost geometries.

### Parameterisation Using Ghost/Parallel Geometries

In some cases, volumes of the tracking geometry do not allow envelopes to be defined. This may be the case with a geometry coming from a CAD system. Since such a geometry is flat, a parallel geometry must be used to define the envelopes.

Another interesting case involves defining an envelope which groups the electromagnetic and hadronic calorimeters of a detector into one volume. This may be useful when parameterising the interaction of charged pions. You will very likely not want electrons to see this envelope, which means that ghost geometries have to be organised by particle flavours.

Using ghost geometries implies some more overhead in the parameterisation mechanism for the particles sensitive to ghosts, since navigation is provided in the ghost geometry by the `G4FastSimulationManagerProcess`. Usually, however, only a few volumes will be placed in this ghost world, so that the geometry computations will remain rather cheap.

For details on how to use fast simulation with the parallel/world geometry please consult *The G4FastSimulationManagerProcess Class*.

### Gflash Parameterisation

This section describes how to use the Gflash parameterisation. Gflash is a concrete implementation based on the equations and parameters of the original Gflash package from H1(hep-ex/0001020, Grindhammer & Peters, see physics manual) and uses the fast simulation facilities of GEANT4 described above. Briefly, whenever a e-/e+ particle enters the calorimeter, it is parameterised if it has a minimum energy and the shower is expected to be contained in the calorimeter (or 'parameterisation envelope'). If this is fulfilled the particle is killed, no secondaries are created, and instead the energy is deposited according to the Gflash equations. Examples, provided in **examples/extended/parametrisation/gflash/**, show how to interface Gflash to your application. The simulation time is measured, so the user can immediately see the speed increase resulting from the use of Gflash.

### Using the Gflash Parameterisation

To use Gflash 'out of the box' the following steps are necessary:

- The user must add the fast simulation process:

```
G4VModularPhysicsList* physicsList = new FTFP_BERT();
G4FastSimulationPhysics* fastSimulationPhysics = new G4FastSimulationPhysics();
fastSimulationPhysics->ActivateFastSimulation("e-");
physicsList->RegisterPhysics( fastSimulationPhysics );
```

- The envelope in which the parameterisation should be performed must be created and specified (below: `G4Region* fRegion`) and the `GFlashShowerModel` must be assigned to this region. Furthermore, the classes `GFlashParticleBounds` (which provides thresholds for the parameterisation like minimal energy etc.), `GflashHitMaker` (a helper class to generate hits in the sensitive detector) and `GFlashHomoShowerParameterisation` (which does the computations) must be constructed and attached to the `GFlashShowerModel`:

```
G4Material* pbWO4 = nistManager->FindOrBuildMaterial("G4_PbWO4");
fFastShowerModel = new GFlashShowerModel("fFastShowerModel", fRegion);
fParameterisation = new GFlashHomoShowerParameterisation(pbWO4);
fFastShowerModel->SetParameterisation(*fParameterisation);
fParticleBounds = new GFlashParticleBounds();
fFastShowerModel->SetParticleBounds(*fParticleBounds);
fHitMaker = new GFlashHitMaker();
fFastShowerModel->SetHitMaker(*fHitMaker);
```

  The user must set the material of the calorimeter, since the computation depends on the material.

- It is mandatory to use `G4VGFlashSensitiveDetector` as (additional) base class for the sensitive detector:

  ```
  class ExGflash1SensitiveDetector: public G4VSensitiveDetector, public␣
  ↪G4VGFlashSensitiveDetector
  ```

  Here it is necessary to implement a separate interface, where the Gflash spots are processed:

  ```
  virtual G4bool ProcessHits(G4GFlashSpot*aSpot,G4TouchableHistory*);
  ```

  A separate interface is used, because the Gflash spots naturally contain less information than the full simulation.

Since the parameters in the Gflash package are taken from fits to full simulations with Geant3, with limited number of materials, and on specific range of electron energy, some re-tuning might be necessary for good agreement with GEANT4 showers. Such parameterisation may be moreover made independent on material (currently atomic number Z) reducing the number of parameters. The tuning procedure is described in hep-ex/0001020 and there is on-going work to automate that procedure and make available in GEANT4. For the moment, the only way to alter the parameters is to provide an implementation of **GVFlashHomoShowerTuning** class and pass it to the class that calculates the showers profiles, etc.:

```
GFlashHomoShowerParameterisation(G4Material * aMat, GVFlashHomoShowerTuning * aPar = 0);
```

For information, there is also a preliminary (still not validated) implementation of a parameterisation for sampling calorimeters. The user must specify the active and passive material, as well as the thickness of the active and passive layer. The sampling structure of the calorimeter is taken into account by using an "effective medium" to compute the shower shape.

All material properties needed are calculated automatically. If tuning is required, the user can pass his own parameter set in the class **GFlashSamplingShowerTuning**. Here the user can also set his calorimeter resolution.

All in all the constructor looks the following:

```
GFlashSamplingShowerParamterisation(G4Material * Mat1, G4Material * Mat2,G4double d1,G4double d2,
GVFlashSamplingShowerTuning * aPar = 0);
```

### 5.2.9 Transportation Process

A Transportation process is required for every particle which is tracked in a simulation. The GEANT4 transportation processes are responsible for several key functions for tracking:

- polling the Geometry Modeller via `G4Navigator` to obtain the distance to the next boundary for uncharged particles or charged particles in a volume / region without an electromagnetic field;
- handing off the tracking of charged particles in an EM field to `G4PropagatorInField` which finds either the endpoint of integration of the equations of motion of the particle or the state of the particle at the location in which it intersects with the next volume boundary;
- updating the time of flight of the particle, using the full step length (not the geometrical step length, which is reduced by multiple scattering.)
- killing tracks which are found to loop inside a (magnetic) field, without making adequate progress after O(thousand) integration steps. A description of this is provided below.

Transportation comes in two flavours:

- the 'standard' `G4Transportation` process, used for most applications, and
- the `G4CoupledTransportation` process, which is activated when multiple geometries are active.

Multiple geometries can be created in order to cope with different use cases:

- when a mass overlay geometry is used to overlap a set of 'top' volume onto a complex existing geometry,
- when the GEANT4 scoring and/or biasing capabilities are activated.

The registration of the relevant Transportation process is handled by the `G4PhysicsListHelper`, which chooses the correct type depending on whether any of the features which require parallel geometries have been used.

In brief there is one main difference between `G4Transportation` and `G4CoupledTransportation`. The `G4Transportation` process uses the `G4Navigator` of the Geant4 Geometry Modeller to obtain the distance to the next boundary along a straight line (for a neutral particle, or a charged particle in zero field). The `G4CoupledTransportation` process uses the `G4PathFinder` class to obtain the shortest length to a boundary amongst the geometries registered for the current particle - in effect multiplexing the different geometries.

In addition the transportation processes estimates the time of flight for the current step. For a neutral particle or a charged particle inside a pure magnetic field, this is estimated from the initial and final velocity of the particle. This taking into account roughly the effect of energy loss from ionisation. Since the full path length is used (rather than the geometrical one) the path lengthening due to multiple scattering is also taken into account.

For a charged particle in an EM field with a non-zero electric component, or a gravity field, the time of flight is calculated taking into account the change in velocity.

For the propagation in an external field, electromagnetic or other, the Transportation processes rely on the capabilities of `G4PropagatorInField` and the integration methods detailed in the subsection ElectroMagnetic Field

Note that the integration currently is done without taking into account either energy loss along the trajectory of motion or multiple scattering, which is applied independently at the endpoint (if it is not on a boundary.)

Further details about the caveats and control of transportation within a magnetic field are given in the Appendix: *Transportation in Magnetic Field - Further Details*.

## 5.3 Particles

### 5.3.1 Basic concepts

There are three levels of classes to describe particles in GEANT4.

**`G4ParticleDefinition`** defines a particle
**`G4DynamicParticle`** describes a particle interacting with materials
**`G4Track`** describes a particle traveling in space and time

`G4ParticleDefinition` aggregates information to characterize a particle's properties, such as name, mass, spin, life time, and decay modes. `G4DynamicParticle` aggregates information to describe the dynamics of particles, such as energy, momentum, polarization, and proper time, as well as "particle definition" information. `G4Track` (see *Tracking*) includes all information necessary for tracking in a detector simulation, such as time, position, and step, as well as "dynamic particle" information.

### 5.3.2 Definition of a particle

There are a large number of elementary particles and nuclei. GEANT4 provides the `G4ParticleDefinition` class to represent particles, and various particles, such as the electron, proton, and gamma have their own classes derived from `G4ParticleDefinition`.

We do not need to make a class in GEANT4 for every kind of particle in the world. There are more than 100 types of particles defined in GEANT4 by default. Which particles should be included, and how to implement them, is determined according to the following criteria. (Of course, the user can define any particles he wants. Please see the **User's Guide: For ToolKit Developers**).

## Particle List in GEANT4

This list includes all particles in GEANT4 and you can see properties of particles such as

- PDG encoding
- mass and width
- electric charge
- spin, isospin and parity
- magnetic moment
- quark contents
- life time and decay modes

Here is a list of particles in GEANT4. This list is generated automatically by using GEANT4 functionality, so listed values are same as those in your GEANT4 application (as far as you do not change source codes).

**Categories**

- gluon / quarks / di-quarks
- leptons
- mesons
- baryons
- ions
- others

## Classification of particles

1. elementary particles which should be tracked in GEANT4 volumes
   All particles that can fly a finite length and interact with materials in detectors are included in this category. In addition, some particles with a very short lifetime are included for user's convenience.
   1. **stable** particles
      Stable means that the particle can not decay, or has a very small possibility to decay in detectors, e.g., gamma, electron, proton, and neutron.
   2. **long life** ($>10^{-14}$sec) particles
      Particles which may travel a finite length, e.g., muon, charged pions.
   3. **short life** particles that decay immediately in GEANT4
      For example, $pi^0$, eta
   4. $K^0$ **system**
      $K^0$ "decays" immediately into $K^0_S$ or $K^0_L$, and then $K^0_S$/ $K^0_L$ decays according to its life time and decay modes.
   5. **optical photon**
      Gamma and optical photon are distinguished in the simulation view, though both are the same particle (photons with different energies). For example, optical photon is used for Cerenkov light and scintillation light.
   6. **geantino/charged geantino**
      Geantino and charged geantino are virtual particles for simulation which do not interact with materials and undertake transportation processes only.
2. nuclei
   Any kinds of nucleus can be used in GEANT4, such as alpha(He-4), uranium-238 and excited states of carbon-14. In addition, GEANT4 provides hyper-nuclei. Nuclei in GEANT4 are divided into two groups from the viewpoint of implementation.
   1. **light nuclei**
      Light nuclei frequently used in simulation, e.g., alpha, deuteron, He3, triton.
   2. **heavy nuclei** (including hyper-nuclei)
      Nuclei other than those defined in the previous category.
   3. **light anti-nuclei**

Light anti-nuclei, for example, anti-alpha.

4. **light hyper-nuclei**
   Light hyper-nuclei and anti-hyper-nuclei, for example hyper-alpha and anti-hyper-alpha

Note that G4ParticleDefinition represents nucleus state and G4DynamicParticle represents atomic state with some nucleus. Both alpha particle with charge of +2e and helium atom with no charge aggregates the same "particle definition" of G4Alpha, but different G4DynamicParticle objects should be assigned to them. (Details can be found below)

3. short-lived particles
   Particles with very short life time decay immediately and are never tracked in the detector geometry. These particles are usually used only inside physics processes to implement some models of interactions. `G4VShortLivedParticle` is provided as the base class for these particles. All classes related to particles in this category can be found in `shortlived` sub-directory under the `particles` directory.
   1. **quarks/di-quarks**: For example, all 6 quarks.
   2. **gluon**
   3. **baryon excited states** with very short life: For example, spin 3/2 baryons and anti-baryons
   4. **meson excited states** with very short life: For example, spin 1 vector bosons

### Implementation of particles

*Single object created in the initialization:* Categories a, b-1

These particles are frequently used for tracking in GEANT4. An individual class is defined for each particle in these categories. The object in each class is unique. The user can get pointers to these objects by using static methods in their own classes. The unique object for each class is created when its static method is called in the "initialization phase:.

*On-the-fly creation:* Category b-2

Ions will travel in a detector geometry and should be tracked, however, the number of ions which may be used for hadronic processes is so huge that ions are dynamically created by requests from processes (and users). Each ion corresponds to one object of the `G4Ions` class. `G4IonTable` class is a dictionary for ions. `G4IonTable::GetIon()` method to create ions on the fly. (`G4IonTable::FIndIon()` method returns pointer to the specified ion. If the ion does not exists, it returns zero without creating any ion.

`G4NucleiPropertiesTableAME03` contains a table of measured mass values of about 3100 stable nuclei (ground states). `G4NucleiPropertiesTheoreticalTable` theoretical mass values of about 8000 nuclei (ground states). `G4IsotopeTable` describes properties of ions (exited energy, decay modes, life time and magnetic moments), which are used to create ions. `G4NuclideTable` is provided as a list of nuclei in GEANT4. It contains about 2900 ground states and 4000 excited states. Users can register his/her `G4IsotopeTable` to the `G4IonTable`.

Processes attached to heavy ions are same as those for `G4GenericIon` class. In other words, you need to create `G4GenericIon` and attach processes to it if you want to use heavy ions.

`G4ParticleGun` can shoot any heavy ions with /gun/ions command after `ion` is selected by /gun/particle command.

*Dynamic creation by processes:* Category c

Particle types in this category are are not created by default, but will only be created by request from processes or directly by users. Each shortlived particle corresponds to one object of a class derived from `G4VshortLivedParticle`, and it will be created dynamically during the `initialization phase`.

### G4ParticleDefinition

The `G4ParticleDefinition` class has "read-only" properties to characterize individual particles, such as name, mass, charge, spin, and so on. These properties are set during initialization of each particle. Methods to get these properties are listed in Table 5.10.

Table 5.10: Methods to get particle properties.

| | |
|---|---|
| `G4String GetParticleName()` | particle name |
| `G4double GetPDGMass()` | mass |
| `G4double GetPDGWidth()` | decay width |
| `G4double GetPDGCharge()` | electric charge |
| `G4double GetPDGSpin()` | spin |
| `G4double GetPDGMagneticMoment()` | magnetic moment (0: not defined or no magnetic moment) |
| `G4int GetPDGiParity()` | parity (0:not defined) |
| `G4int GetPDGiConjugation()` | charge conjugation (0:not defined) |
| `G4double GetPDGIsospin()` | iso-spin |
| `G4double GetPDGIsospin3()` | 3$^{rd}$-component of iso-spin |
| `G4int GetPDGiGParity()` | G-parity (0:not defined) |
| `G4String GetParticleType()` | particle type |
| `G4String GetParticleSubType()` | particle sub-type |
| `G4int GetLeptonNumber()` | lepton number |
| `G4int GetBaryonNumber()` | baryon number |
| `G4int GetPDGEncoding()` | particle encoding number by PDG |
| `G4int GetAntiPDGEncoding()` | encoding for anti-particle of this particle |

Table 5.11 shows the methods of `G4ParticleDefinition` for getting information about decay modes and the life time of the particle.

Table 5.11: Methods to get particle decay modes and life time.

| | |
|---|---|
| `G4bool GetPDGStable()` | stable flag |
| `G4double GetPDGLifeTime()` | life time |
| `G4DecayTable* GetDecayTable()` | decay table |

Users can modify these properties, though the other properties listed above can not be change without rebuilding the libraries.

Each particle has its own `G4ProcessManger` object that manages a list of processes applicable to the particle.(see *Managing Processes*)

## 5.3.3 Dynamic particle

The `G4DynamicParticle` class has kinematics information for the particle and is used for describing the dynamics of physics processes. The properties in `G4DynamicParticle` are listed in the Table 5.12.

Table 5.12: Methods to set/get values.

| | |
|---|---|
| `G4double theDynamicalMass` | dynamical mass |
| `G4ThreeVector theMomentumDirection` | normalized momentum vector |
| `G4ParticleDefinition* theParticleDefinition` | definition of particle |
| `G4double theDynamicalSpin` | dynamical spin (i.e. total angular momentum as a ion/atom) |
| `G4ThreeVector thePolarization` | polarization vector |
| `G4double theMagneticMoment` | dynamical magnetic moment (i.e. total magnetic moment as a ion/atom ) |
| `G4double theKineticEnergy` | kinetic energy |
| `G4double theProperTime` | proper time |
| `G4double theDynamicalCharge` | dynamical electric charge (i.e. total electric charge as a ion/atom ) |
| `G4ElectronOccupancy* theElectronOccupancy` | electron orbits for ions |

Here, the dynamical mass is defined as the mass for the dynamic particle. For most cases, it is same as the mass defined in `G4ParticleDefinition` class ( i.e. mass value given by `GetPDGMass()` method). However, there are two exceptions.

- resonance particle
- ions

Resonance particles have large mass width and the total energy of decay products at the center of mass system can be different event by event.

As for ions, `G4ParticleDefintion` defines a nucleus and `G4DynamicParticle` defines an atom. `G4ElectronOccupancy` describes state of orbital electrons. So, the dynamic mass can be different from the PDG mass by the mass of electrons (and their binding energy). In addition, the dynamical charge, spin and magnetic moment are those of the atom/ion (i.e. including nucleus and orbit electrons).

Decay products of heavy flavor particles are given in many event generators. In such cases, `G4VPrimaryGenerator` sets this information in `*thePreAssignedDecayProducts`. In addition, decay time of the particle can be set arbitrarily time by using `PreAssignedDecayProperTime`.

## 5.4 Production Threshold versus Tracking Cut

### 5.4.1 General considerations

We have to fulfill two contradictory requirements. It is the responsibility of each individual **process** to produce secondary particles according to its own capabilities. On the other hand, it is only the GEANT4 kernel (i.e., tracking) which can ensure an overall coherence of the simulation.

The general principles in GEANT4 are the following:

1. Each **process** has its intrinsic limit(s) to produce secondary particles.
2. All particles produced (and accepted) will be tracked up to **zero range**.
3. Each **particle** has a suggested cut in range (which is converted to energy for all materials), and defined via a `SetCut()` method (see *Range Cuts*).

Points 1 and 2 imply that the cut associated with the **particle** is a (recommended) **production** threshold of secondary particles.

### 5.4.2 Set production threshold (`SetCut` methods)

As already mentioned, each kind of particle has a suggested production threshold. Some of the processes will not use this threshold (e.g., decay), while other processes will use it as a default value for their intrinsic limits (e.g., ionisation and bremsstrahlung).

See *Range Cuts* to see how to set the production threshold.

### 5.4.3 Apply cut

The `DoIt` methods of each process can produce secondary particles. Two cases can happen:

- a process sets its intrinsic limit greater than or equal to the recommended production threshold. OK. Nothing has to be done (nothing can be done !).
- a process sets its intrinsic limit smaller than the production threshold (for instance 0).

The list of secondaries is sent to the *SteppingManager* via a *ParticleChange* object.

*Before* being recopied to the temporary stack for later tracking, the particles below the production threshold will be kept or deleted according to the safe mechanism explained hereafter.

- The *ParticleDefinition* (or *ParticleWithCuts*) has a Boolean data member: `ApplyCut`.
- `ApplyCut` is OFF: do nothing. All the secondaries are stacked (and then tracked later on), regardless of their initial energy. The GEANT4 kernel respects the best that the physics can do, but neglects the overall coherence and the efficiency. Energy conservation is respected as far as the processes know how to handle correctly the particles they produced! This is the main used during GEANT4 tracking.
- `ApplyCut` in ON: this feature is not normally used but is potentially available; the *TrackingManager* checks the range of each secondary against the production threshold and against the safety. The particle is stacked if `range > min(cut,safety)`.
  - If not, check if the process has nevertheless set the flag "good for tracking" and then stack it (see *Why produce secondaries below threshold in some processes?* below for the explanation of the `GoodForTracking` flag).
  - If not, recuperate its kinetic energy in the `localEnergyDeposit`, and set `tkin=0`.
  - Then check in the *ProcessManager* if the vector of *ProcessAtRest* is not empty. If yes, stack the particle for performing the "Action At Rest" later. If not, and only in this case, abandon this secondary.

With this sophisticated mechanism we have the global cut that we wanted, but with energy conservation, and we respect boundary constraint (safety) and the wishes of the processes (via "good for tracking"). Note, that for electromagnetic processes for gamma incident a specific `ApplyCut` option is used which guarantees energy balance and is more efficient because secondary tracks are not produced at all.

### 5.4.4 Why produce secondaries below threshold in some processes?

A process may have good reasons to produce particles below the recommended threshold:

- checking the range of the secondary versus geometrical quantities like safety may allow one to realize the possibility that the produced particle, even below threshold, will reach a sensitive part of the detector;
- another example is the gamma conversion: the positron is always produced, even at zero energy, for further annihilation;
- if a process is rare there is not practical reason make it complicate checking cut value.

These secondary particles are sent to the "Stepping Manager" with a flag `GoodForTracking` to pass the filter explained in the previous section (even when `ApplyCut` is ON).

### 5.4.5 Cuts in stopping range or in energy?

The cuts in stopping range allow one to say that the energy has been released at the correct space position, limiting the approximation within a given distance. On the contrary, cuts in energy imply accuracies of the energy depositions which depend on the material.

### 5.4.6 Summary

In summary, we do not have tracking cuts; we only have production thresholds in range. All particles produced and accepted are tracked up to zero range.

It must be clear that the overall coherency that we provide cannot go beyond the capability of processes to produce particles down to the recommended threshold.

In other words a process can produce the secondaries down to the recommended threshold, and by interrogating the geometry, or by realizing when mass-to-energy conversion can occur, recognize when particles below the threshold have to be produced.

### 5.4.7 Special tracking cuts

One may need to cut given particle types in given volumes for optimisation reasons. This decision is under user control, and can happen for particles during tracking as well.

The user must be able to apply these special cuts only for the desired particles and in the desired volumes, without introducing an overhead for all the rest.

The approach is as follows:

- special user cuts are registered in the *UserLimits* class (or its descendant), which is associated with the logical volume class.
  The current default list is:
  - max allowed step size
  - max total track length
  - max total time of flight
  - min kinetic energy
  - min remaining range

  The user can instantiate a *UserLimits* object only for the desired logical volumes and do the association.
  The first item (max step size) is automatically taken into account by the G4 kernel while the others items must be managed by the user, as explained below.
  **Example**(see basic/B2/B2a or B2b): in the Tracker region, in order to force the step size not to exceed one half of the Tracker chamber thickness (`chamberWidth`), it is enough to put the following code in `B2a::DetectorConstruction::DefineVolumes()`:

  ```
  G4double maxStep = 0.5*chamberWidth;
  fStepLimit = new G4UserLimits(maxStep);
  trackerLV->SetUserLimits(fStepLimit);
  ```

  and in `PhysicsList`, the process `G4StepLimiter` needs to be attached to each particle's process manager where step limitation in the Tracker region is required:

  ```
  // Step limitation seen as a process
  G4StepLimiter* stepLimiter = new G4StepLimiter();
  pmanager->AddDiscreteProcess(StepLimiter);
  ```

  If a provided GEANT4 physics list is used, as FTFP_BERT in B2 example, then the `G4StepLimiterPhysics`, which will take care of attaching the `G4StepLimiter` process to all particles, can be added to the physics list in the `main()` function:

```
G4VModularPhysicsList* physicsList = new FTFP_BERT;
physicsList->RegisterPhysics(new G4StepLimiterPhysics());
runManager->SetUserInitialization(physicsList);
```

The `G4UserLimits` class is in `source/global/management`.

- Concerning the others cuts, the user must define dedicated process(es). He registers this process (or its descendant) only for the desired particles in their process manager. He can apply his cuts in the `DoIt` of this process, since, via `G4Track`, he can access the logical volume and *UserLimits*.

  An example of such process (called *UserSpecialCuts*) is provided in the repository, but not inserted in any process manager of any particle.

  **Example: neutrons.** One may need to abandon the tracking of neutrons after a given time of flight (or a charged particle in a magnetic field after a given total track length … etc …).

  Example(see basic/B2/B2a or B2b): in the Tracker region, in order to force the total time of flight of the neutrons not to exceed 10 milliseconds, put the following code in `B2a::DetectorConstruction::DefineVolumes()`:

```
G4double maxTime = 10*ms;
fStepLimit = new G4UserLimits(DBL_MAX,DBL_MAX,maxTime);
trackerLV->SetUserLimits(fStepLimit);
```

  and put the following code in a physics list:

```
G4ProcessManager* pmanager = G4Neutron::Neutron->GetProcessManager();
pmanager->AddProcess(new G4UserSpecialCuts(),-1,-1,1);
```

  If a provided GEANT4 physics list is used, then a `SpecialCutsBuilder` class can be defined in a similar way as `G4StepLimiterPhysics` and added to the physics list in the `main()` function:

```
G4VModularPhysicsList* physicsList = new FTFP_BERT;
physicsList->RegisterPhysics(new SpecialCutsBuilder());
runManager->SetUserInitialization(physicsList);
```

  (The default `G4UserSpecialCuts` class is in `source/processes/transportation`.)

## 5.5 Cuts per Region

### 5.5.1 General Concepts

Beginning with GEANT4 version 5.1, the concept of a region has been defined for use in geometrical descriptions. Details about regions and how to use them are available in *Sub-detector Regions*. As an example, suppose a user defines three regions, corresponding to the tracking volume, the calorimeter and the bulk structure of a detector. For performance reasons, the user may not be interested in the detailed development of electromagnetic showers in the insensitive bulk structure, but wishes to maintain the best possible accuracy in the tracking region. In such a use case, GEANT4 allows the user to set different production thresholds ("cuts") for each geometrical region. This ability, referred to as "cuts per region", is also a new feature provided by the GEANT4 5.1 release. The general concepts of production thresholds were presented in the *Production Threshold versus Tracking Cut*.

Please note that this new feature is intended only for users who

1. are simulating the most complex geometries, such as an LHC detector, and
2. are experienced in simulating electromagnetic showers in matter.

We strongly recommend that results generated with this new feature be compared with results using the same geometry and uniform production thresholds. Setting completely different cut values for individual regions may break the coherent and comprehensive accuracy of the simulation. Therefore cut values should be carefully optimized, based on a comparison with results obtained using uniform cuts.

### 5.5.2 Default Region

The world volume is treated as a region by default. A G4Region object is automatically assigned to the world volume and is referred to as the "default region". The production cuts for this region are the defaults which are defined in the *UserPhysicsList*. Unless the user defines different cut values for other regions, the cuts in the default region will be used for the entire geometry.

Please note that the default region and its default production cuts are created and set automatically by G4RunManager. The user is **not** allowed to set a region to the world volume, **nor** to assign other production cuts to the default region.

### 5.5.3 Assigning Production Cuts to a Region

In the SetCuts() method of the user's physics list, the user must first define the default cuts. Then a G4ProductionCuts object must be created and initialized with the cut value desired for a given region. This object must in turn be assigned to the region object, which can be accessed by name from the G4RegionStore. An example SetCuts() code follows.

Listing 5.12: Setting production cuts to a region

```
void MyPhysicsList::SetCuts()
{
  // default production thresholds for the world volume
  SetCutsWithDefault();

  // Production thresholds for detector regions
  G4Region* region;
  G4String regName;
  G4ProductionCuts* cuts;

  regName = "tracker";
  region = G4RegionStore::GetInstance()->GetRegion(regName);
  cuts = new G4ProductionCuts;
  cuts->SetProductionCut(0.01*mm); // same cuts for gamma, proton, e- and e+
  region->SetProductionCuts(cuts);

  regName = "calorimeter";
  region = G4RegionStore::GetInstance()->GetRegion(regName);
  cuts = new G4ProductionCuts;
  cuts->SetProductionCut(0.01*mm,G4ProductionCuts::GetIndex("gamma"));
  cuts->SetProductionCut(0.1*mm,G4ProductionCuts::GetIndex("e-"));
  cuts->SetProductionCut(0.1*mm,G4ProductionCuts::GetIndex("e+"));
  cuts->SetProductionCut(0.1*mm,G4ProductionCuts::GetIndex("proton"));
  region->SetProductionCuts(cuts);
}
```

## 5.6 Physics Table

### 5.6.1 General Concepts

In GEANT4, physics processes use many tables of cross sections, energy losses and other physics values. Before the execution of an event loop, PreparePhysicsTable() and BuildPhysicsTable() methods of G4VProcess are invoked for all processes and as a part of initialisation procedure cross section tables are prepared. Energy loss processes calculate cross section and/or energy loss values for each pair of material and production cut value used in geometry for a give run. A change in production cut values therefore require these cross sections to be re-calculated. Cross sections for hadronic processes and gamma processes do not depend on the production cut but sampling of final state may depend on cuts, so full re-initialisation is performed.

The `G4PhysicsTable` class is used to handle cross section tables. `G4PhysicsTable` is a collection of instances of `G4PhysicsVector` (and derived classes), each of which has cross section values for a particle within a given energy range traveling in a material. By default the linear interpolation is used, alternatively spline may be used if the flag of spline is activated by *SetSpline* method of the `G4PhysicsVector`

### 5.6.2 Material-Cuts Couple

Users can assign different production cuts to different regions (see *Cuts per Region*). This means that if the same material is used in regions with different cut values, the processes need to prepare several different cross sections for that material.

The `G4ProductionCutsTable` has `G4MaterialCutsCouple` objects, each of which consists of a material paired with a cut value. These `G4MaterialCutsCouple`s are numbered with an index which is the same as the index of a `G4PhysicsVector` for the corresponding `G4MaterialCutsCouple`in the `G4PhysicsTable`. The list of *MaterialCutsCouple*s used in the current geometry setup is updated before starting the event loop in each run.

### 5.6.3 File I/O for the Physics Table

Calculated physics tables for electromagnetic processes can be stored in files. The user may thus eliminate the time required for the calculation of physics tables by retrieving them from the files.

Using the built-in user command "**storePhysicsTable**" (see *Built-in Commands*), stores physics tables in files. Information on materials and cuts defined in the current geometry setup are stored together with physics tables because calculated values in the physics tables depend on *MaterialCutsCouple*. Note that physics tables are calculated before the event loop, not in the initialization phase. So, at least one event must be executed before using the "**storePhysicsTable**" command.

Calculated physics tables can be retrieved from files by using the "**retrievePhysicsTable**" command. Materials and cuts from files are compared with those defined in the current geometry setup, and only physics vectors corresponding to the *MaterialCutsCouple*s used in the current setup are restored. Note that nothing happens just after the "**retrievePhysicsTable**" command is issued. Restoration of physics tables will be executed in parallel with the calculation of physics tables.

### 5.6.4 Building the Physics Table

In the `G4RunManagerKernel::RunInitialization()` method, after the list of *MaterialCutsCouple*s is updated, the `G4VUserPhysicsList::BuildPhysicsTable()` method is invoked to build physics tables for all processes.

Initially, the `G4VProcess::PreparePhysicsTable()` method is invoked. Each process creates `G4PhysicsTable` objects as necessary. It then checks whether the *MaterialCutsCouple*s have been modified after a run to determine if the corresponding physics vectors can be used in the next run or need to be re-calculated.

Next, the `G4VProcess::RetrievePhysicsTable()` method is invoked if the `G4VUserPhysicsList::fRetrievePhysicsTable` flag is asserted. After checking materials and cuts in files, physics vectors corresponding to the *MaterialCutsCouple*s used in the current setup are restored.

Finally, the `G4VProcess::BuildPhysicsTable()` method is invoked and only physics vectors which need to be re-calculated are built.

At the end of program `G4PhysicsTable` should be deleted. Before deletion of a table it should be cleaned up using the method `G4PhysicsTable::clearAndDestroy()`. This method should be called in a middle of the run if an old table is removed and a new one is created.

## 5.7 User Limits

### 5.7.1 General Concepts

The user can define artificial limits affecting to the GEANT4 tracking.

```
G4UserLimits(G4double uStepMax = DBL_MAX,
             G4double uTrakMax = DBL_MAX,
             G4double uTimeMax = DBL_MAX,
             G4double uEkinMin = 0.,
             G4double uRangMin = 0. );
```

where:

| | |
|---|---|
| `uStepMax` | Maximum step length |
| `uTrakMax` | Maximum total track length |
| `uTimeMax` | Maximum global time for a track |
| `uEkinMin` | Minimum remaining kinetic energy for a track |
| `uRangMin` | Minimum remaining range for a track |

Note that `uStepMax` is affecting to each step, while all other limits are affecting to a track.

The user can assign `G4UserLimits` to logical volume and/or to a region. User limits assigned to logical volume do not propagate to daughter volumes, while User limits assigned to region propagate to daughter volumes unless daughters belong to another region. If both logical volume and associated region have user limits, those of logical volume win.

A G4UserLimits object must be instantiated for the duration of whatever logical volume or region to which it is assigned. It is the responsibility of the user's code to delete the object *after* the assigned volume(s)/region(s) have been deleted.

### 5.7.2 Processes co-working with G4UserLimits

In addition to instantiating `G4UserLimits` and setting it to logical volume or region, the user has to assign the following process(es) to particle types he/she wants to affect. If none of these processes is assigned, that kind of particle is not affected by `G4UserLimits`.

**Limitation to step (`uStepMax`)** `G4StepLimiter` process must be defined to affected particle types. This process limits a step, but it does not kill a track.

**Limitations to track (`uTrakMax, uTimeMax, uEkinMin, uRangMin`)** `G4UserSpecialCuts` process must be defined to affected particle types. This process limits a step and kills the track when the track comes to one of these limits. Step limitation occurs only for the final step.

Example of `G4UserLimits` can be found in examples/basic/B2 : see `B2a::DetectorConstruction` (or `B2b::DetectorConstruction`). The `G4StepLimiter` process is added in the GEANT4 physics list via the `G4StepLimiterPhysics` class in the `main()` function in `exampleB2a.cc` (or `exampleB2b.cc`).

## 5.8 Track Error Propagation

The error propagation package serves to propagate one particle together with its error from a given trajectory state until a user-defined target is reached (a surface, a volume, a given track length,. . . ).

### 5.8.1 Physics

The error propagator package computes the average trajectory that a particle would follow. This means that the physics list must have the following characteristics:

- No multiple scattering
- No random fluctuations for energy loss
- No creation of secondary tracks
- No hadronic processes

It has also to be taken into account that when the propagation is done backwards (in the direction opposed to the one the original track traveled) the energy loss has to be changed into an energy gain.

All this is done in the `G4ErrorPhysicsList` class, that is automatically set by `G4ErrorPropagatorManager` as the GEANT4 physics list. It sets `G4ErrorEnergyLoss` as unique electromagnetic process. This process uses the GEANT4 class `G4EnergyLossForExtrapolator` to compute the average energy loss for forwards or backwards propagation. To avoid getting too different energy loss calculation when the propagation is done forwards (when the energy at the beginning of the step is used) or backwards (when the energy at the end of the step is used, always smaller than at the beginning) `G4ErrorEnergyLoss` computes once the energy loss and then replaces the original energy loss by subtracting/adding half of this value (what is approximately the same as computing the energy loss with the energy at the middle of the step). In this way, a better calculation of the energy loss is obtained with a minimal impact on the total CPU time.

The user may use his/her own physics list instead of `G4ErrorPhysicsList`. As it is not needed to define a physics list when running this package, the user may have not realized that somewhere else in his/her application it has been defined; therefore a warning will be sent to advert the user that he is using a physics list different to `G4ErrorPhysicsList`. If a new physics list is used, it should also initialize the `G4ErrorMessenger` with the classes that serve to limit the step:

```
G4ErrorEnergyLoss* eLossProcess = new G4ErrorEnergyLoss;
G4ErrorStepLengthLimitProcess* stepLengthLimitProcess = new G4ErrorStepLengthLimitProcess;
G4ErrorMagFieldLimitProcess* magFieldLimitProcess = new G4ErrorMagFieldLimitProcess;
new G4ErrorMessenger( stepLengthLimitProcess, magFieldLimitProcess, eLossProcess );
```

To ease the use of this package in the reconstruction code, the physics list, whether `G4ErrorPhysicsList` or the user's one, will be automatically initialized before starting the track propagation if it has not been done by the user.

### 5.8.2 Trajectory state

The user has to provide the particle trajectory state at the initial point. To do this it has to create an object of one of the children classes of `G4ErrorTrajState`, providing:

- Particle type
- Position
- Momentum
- Trajectory error matrix

```
G4ErrorTrajState( const G4String& partType,
                  const G4Point3D& pos,
                  const G4Vector3D& mom,
                  const G4ErrorTrajErr& errmat = G4ErrorTrajErr(5,0) );
```

A particle trajectory is characterized by five independent variables as a function of one parameter (e.g. the path length). Among the five variables, one is related to the curvature (to the absolute value of the momentum), two are related to the direction of the particle and the other two are related to the spatial location.

There are two possible representations of these five parameters in the error propagator package: as a free trajectory state, class `G4ErrorTrajStateFree`, or as a trajectory state on a surface, class `G4ErrorTrajStateonSurface`.

### Free trajectory state

In the free trajectory state representation the five trajectory parameters are

- G4double fInvP
- G4double fLambda
- G4double fPhi
- G4double fYPerp
- G4double fZPerp

where `fInvP` is the inverse of the momentum. `fLambda` and `fPhi` are the dip and azimuthal angles related to the momentum components in the following way:

`p\_x = p cos(lambda) cos(phi) p\_y = p cos(lambda) sin(phi) p\_z = p sin(lambda)`, that is, `lambda = 90 - theta`, where `theta` is the usual angle with respect to the Z axis.

`fYperp` and `fZperp` are the coordinates of the trajectory in a local orthonormal reference frame with the X axis along the particle direction, the Y axis being parallel to the X-Y plane (obtained by the vectorial product of the global Z axis and the momentum).

### Trajectory state on a surface

In the trajectory state on a surface representation the five trajectory parameters are

- G4double fInvP
- G4double fPV
- G4double fPW
- G4double fV
- G4double fW

where `fInvP` is the inverse of the momentum; `fPV` and `fPW` are the momentum components in an orthonormal coordinate system with axis U, V and W; `fV` and `fW` are the position components on this coordinate system.

For this representation the user has to provide the plane where the parameters are calculated. This can be done by providing two vectors, V and W, contained in the plane:

```
G4ErrorSurfaceTrajState( const G4String& partType,
                         const G4Point3D& pos,
                         const G4Vector3D& mom,
                         const G4Vector3D& vecV,
                         const G4Vector3D& vecW,
                         const G4ErrorTrajErr& errmat = G4ErrorTrajErr(5,0) );
```

or by providing a plane

```
G4ErrorSurfaceTrajState( const G4String& partType,
                         const G4Point3D& pos,
                         const G4Vector3D& mom,
                         const G4Plane3D& plane,
                         const G4ErrorTrajErr& errmat = G4ErrorTrajErr(5,0) );
```

In this second case the vector V is calculated as the vector in the plane perpendicular to the global vector X (if the plane normal is equal to X, Z is used instead) and W is calculated as the vector in the plane perpendicular to V.

### 5.8.3 Trajectory state error

The 5X5 error matrix should also be provided at the creation of the trajectory state as a `G4ErrorTrajErr` object. If it is not provided a default object will be created filled with null values.

Currently the `G4ErrorTrajErr` is a `G4ErrorSymMatrix`, a simplified version of `CLHEP HepSymMatrix`.

The error matrix is given in units of GeV and cm. Therefore you should do the conversion if your code is using other units.

### 5.8.4 Targets

The user has to define up to where the propagation must be done: the target. The target can be a surface `G4ErrorSurfaceTarget`, which is not part of the GEANT4 geometry. It can also be the surface of a GEANT4 volume `G4ErrorGeomVolumeTarget`, so that the particle will be stopped when it enters this volume. Or it can be that the particle is stopped when a certain track length is reached, by implementing a `G4ErrorTrackLengthTarget`.

#### Surface target

When the user chooses a `G4ErrorSurfaceTarget` as target, the track is propagated until the surface is reached. This surface is not part of GEANT4 geometry, but usually traverses many GEANT4 volumes. The class `G4ErrorNavigator` takes care of the double navigation: for each step the step length is calculated as the minimum of the step length in the full geometry (up to a GEANT4 volume surface) and the distance to the user-defined surface. To do it, `G4ErrorNavigator` inherits from `G4Navigator` and overwrites the methods `ComputeStep()` and `ComputeSafety()`. Two types of surface are currently supported (more types could be easily implemented at user request): plane and cylindrical.

#### Plane surface target

`G4ErrorPlaneSurfaceTarget` implements an infinite plane surface. The surface can be given as the four coefficients of the plane equation `ax+by+cz+d = 0`:

```
G4ErrorPlaneSurfaceTarget(G4double a=0,
                          G4double b=0,
                          G4double c=0,
                          G4double d=0);
```

or as the normal to the plane and a point contained in it:

```
G4ErrorPlaneSurfaceTarget(const G4Normal3D &n,
                          const G4Point3D &p);
```

or as three points contained in it:

```
G4ErrorPlaneSurfaceTarget(const G4Point3D &p1,
                          const G4Point3D &p2,
                          const G4Point3D &p3);
```

#### Cylindrical surface target

G4ErrorCylSurfaceTarget implements an infinite-length cylindrical surface (a cylinder without end-caps). The surface can be given as the radius, the translation and the rotation

```
G4ErrorCylSurfaceTarget( const G4double& radius,
                         const G4ThreeVector& trans=G4ThreeVector(),
                         const G4RotationMatrix& rotm=G4RotationMatrix() );
```

or as the radius and the affine transformation

```
G4ErrorCylSurfaceTarget( const G4double& radius,
                         const G4AffineTransform& trans );
```

#### Geometry volume target

When the user chooses a `G4ErrorGeomVolumeTarget` as target, the track is propagated until the surface of a GEANT4 volume is reached. User can choose if the track will be stopped only when the track enters the volume, only when the track exits the volume or in both cases.

The object has to be instantiated giving the name of a logical volume existing in the geometry:

```
G4ErrorGeomVolumeTarget( const G4String& name );
```

#### Track Length target

When the user chooses a `G4ErrorTrackLengthTarget` as target, the track is propagated until the given track length is reached.

The object has to be instantiated giving the value of the track length:

```
G4ErrorTrackLengthTarget(const G4double maxTrkLength );
```

It is implemented as a `G4VDiscreteProcess` and it limits the step in `PostStepGetPhysicalInteractionLength`. To ease its use, the process is registered to all particles in the constructor.

### 5.8.5 Managing the track propagation

The user needs to propagate just one track, so there is no need of run and events. neither of `G4VPrimaryGeneratorAction`. `G4ErrorPropagator` creates a track from the information given in the `G4ErrorTrajState` and manages the step propagation. The propagation is done by the standard GEANT4 methods, invoking `G4SteppingManager::Stepping()` to propagate each step.

After one step is propagated, `G4ErrorPropagator` takes cares of propagating the track errors for this step, what is done by `G4ErrorTrajStateFree::PropagateError()`. The equations of error propagation are only implemented in the representation of `G4ErrorTrajStateFree`. Therefore if the user has provided instead a `G4ErrorTrajStateOnSurface` object, it will be transformed into a `G4ErrorTrajStateFree` at the beginning of tracking, and at the end it is converted back into `G4ErrorTrajStateOnSurface` on the target surface (on the normal plane to the surface at the final point).

The user `G4VUserTrackingAction::PreUserTrackingAction( const G4Track* )` and `G4VUserTrackingAction::PreUserTrackingAction( const G4Track* )` are also invoked at the beginning and at the end of the track propagation.

G4ErrorPropagator stops the tracking when one of the three conditions is true:

- Energy is exhausted
- World boundary is reached
- User-defined target is reached

In case the defined target is not reached, G4ErrorPropagator::Propagate() returns a negative value.

The propagation of a trajectory state until a user defined target can be done by invoking the method of G4ErrorPropagatorManager

```
G4int Propagate( G4ErrorTrajState* currentTS, const G4ErrorTarget* target,
                 G4ErrorMode mode = G4ErrorMode_PropForwards );
```

You can get the pointer to the only instance of G4ErrorPropagatorManager with

```
G4ErrorPropagatorManager* g4emgr = G4ErrorPropagatorManager::GetErrorPropagatorManager();
```

Another possibility is to invoke the propagation step by step, returning control to the user after each step. This can be done with the method

```
G4int PropagateOneStep( G4ErrorTrajState* currentTS,
                        G4ErrorMode mode = G4ErrorMode_PropForwards );
```

In this case you should register the target first with the command

```
G4ErrorPropagatorData::GetG4ErrorPropagatorData()->SetTarget( theG4eTarget );
```

### Error propagation

As in the GEANT3-based GEANE package, the error propagation is based on the equations of the European Muon Collaboration, that take into account:

- Error from curved trajectory in magnetic field
- Error from multiple scattering
- Error from ionization

The formulas assume propagation along an helix. This means that it is necessary to make steps small enough to assure magnetic field constantness and not too big energy loss.

## 5.8.6 Limiting the step

There are three ways to limit the step. The first one is by using a fixed length value. This can be set by invoking the user command:

```
G4UImanager::GetUIpointer()->ApplyCommand("/geant4e/limits/stepLength MY_VALUE MY_UNIT");
```

The second one is by setting the maximum percentage of energy loss in the step (or energy gain is propagation is backwards). This can be set by invoking the user command:

```
G4UImanager::GetUIpointer()->ApplyCommand("/geant4e/limits/energyLoss MY_VALUE");
```

The last one is by setting the maximum difference between the value of the magnetic field at the beginning and at the end of the step. Indeed what is limited is the curvature, or exactly the value of the magnetic field divided by the value of the momentum transverse to the field. This can be set by invoking the user command:

```
G4UImanager::GetUIpointer()->ApplyCommand("/geant4e/limits/magField MY_VALUE");
```

The classes that limit the step are implemented as GEANT4 processes. Therefore, the invocation of the above-mentioned commands should only be done after the initialization (for example after `G4ErrorPropagatorManager::InitGeant4e()`.

## 5.9 Exotic Physics

The GEANT4 toolkit has recently been extended to include "exotic physics". This covers the area of phonon propagation and crystal channelling. These two domains are applicable for Dark Matter experiments (phonon excitation) and beam extraction and collimation (crystal channelling). The framework within GEANT4 is similar in that a macroscopic periodic crystal lattice is required for both and wave functions are propagated within the medium (rather than discrete particles as in the case of conventional GEANT4). Contained here is a brief description of how to modify a GEANT4 application to include the crystal as both a material and a geometry (plane orientations).

### 5.9.1 Physics

For a more complete description and understanding the user is referred to the extended examples category "exotic-physics" and the references therein.

### 5.9.2 Material

The implementation of solid-state processes in GEANT4 requires the addition of two important features, the crystal unit cell with all its parameters and the support for other data required by the processes. The extended data for a material is stored in a class derived from the virtual class `G4VMaterialExtension`. The `G4ExtenededMaterial` class collects the pointers to concrete instances of `G4VMaterialExtension`. The `G4CrystalExtension` class is a derived class of `G4VMaterialExtension` and collects information on the physics properties of a perfect crystal. In particular, the class contains a pointer to a `G4CrystalUnitCell` object, the elasticity tensor, a map of `G4CrystalAtomBase` objects associated with a `G4Element` and a vector of `G4AtomicBond`. The `G4CrystalUnitCell` class collects information on the mathematical description of the crystal unit cell, i.e. the sizes and the angles of the unit cell, the space group, the Bravais lattice and the lattice system, and methods for the calculation of the volume in the direct and reciprocal space, the spacing between two planes, the angle between two planes, and for the filling of the reduced elasticity tensor. The `G4CrystalExtension` constructor takes as argument a pointer to a `G4Material` object and has to be registered to the `G4ExtendedMaterial` to which it is attached. The `G4CrystalAtomBase` class stores the position of atoms in the crystal unit cell. Since the `G4CrystalAtomBase` class is mapped to a `G4Element` in the `G4CrystalMaterial`, each `G4Element` should have an associated `G4CrystalAtomBase`. The `G4AtomicBond` class contains information on the atomic bond in the crystal. For each instance of the class two `G4Elements` have to be specified as well as the atom number in the `G4CrystalAtomBase` associated to the `G4Element`.

### 5.9.3 Geometry

The `G4LogicalCrystalVolume` accepts only a pointer to a `G4CrystalExtension` in its constructor and stores the definition of the orientation of the crystalline structure with respect to the solid to which it is attached. By convention, the crystal $< 100 >$ direction is by default set parallel to the {[1,0,0]} direction in the GEANT4 reference system, and the $< 010 >$ axis lays on the plane which contains the [1,0,0] and [0,1,0] directions in the GEANT4 reference system.

CHAPTER

SIX

# USER ACTIONS

## 6.1 User Actions

GEANT4 has two user initialization classes and one user action class whose methods the user must override in order to implement a simulation. They require the user to define the detector, specify the physics to be used, and define how initial particles are to be generated. These classes are described in *Mandatory User Actions and Initializations*.

Additionally, users may define any of several optional user actions, to collect data during event generation from steps, tracks, or whole events, to accumulate data during runs, or to modify the state of new tracks as they are created. These user actions are described in *Optional User Actions*.

To support the accumulation of data in the actions mentioned above, users may define subclasses for some of the container objects used during event generation and tracking. These are described in *User Information Classes*.

## 6.2 Mandatory User Actions and Initializations

Three user initialization class objects are registered with the run manager (*Manage the run procedures*) in the user's `main()` program, which takes ownership. The user must not delete these objects directly, and they must be created using 'new'. Within the `G4UserActionInitialization` class ( *User Action Initialization*), the user must instantiate and register a concrete `G4VUserPrimaryGeneratorAction` subclass, which generates the primary particles for each event.

### 6.2.1 G4VUserDetectorConstruction

Listing 6.1: `G4VUserDetectorConstruction`

```
class G4VUserDetectorConstruction
{
  public:
    G4VUserDetectorConstruction();
    virtual ~G4VUserDetectorConstruction();

  public:
    virtual G4VPhysicalVolume* Construct() = 0;
    virtual void ConstructSDandField() = 0;
};
```

In the `Construct()` method, material and geometry has to be described. Detailed discussions on material and geometry are given in *How to Specify Materials in the Detector* and *How to Define a Detector Geometry*. Detector sensitivity and electromagnetic field should be defined in `ConstructSDandField()`, as objects defined in this method are thread-local if they are used in multi-threaded mode. Detailed discussions on Detector sensitivity and electromagnetic field are given in *Hits* and *Electromagnetic Field*.

### 6.2.2 Physics Lists

The concept of a physics list arises from the fact that GEANT4 can not offer a single modeling algorithm to cover the entire energy domain from zero to the TeV scale, for all known processes and particles. Instead, a combination of ideas and approaches is typically used to perform a simulation task.

A schematic view of the GEANT4 modeling of the processes of particle passage through matter may be presented as follows:

- Physics Model = final state generator
- Physics Process = cross section + model
- Physics List = list of processes for each particle

The "patchwork" concept is especially true in the GEANT4 hadronic physics domain: models are valid only over finite energy ranges, and there maybe competing models in the same range or one model maybe working better than the other for a specific group of particles, while its competitor may be better for other species. For this reason models have to be combined to cover the large energy range; every two adjacent models may have an overlap in their validity range.

#### `G4VUserPhysicsList`

This is an abstract class for constructing particles and processes. An introduction into the concept of the GEANT4 Physics List and the GEANT4 Physics Processes is also given in *How to Specify Physics Processes* and further in *Physics Processes*.

While the fabrication of a physics list is, in principle, a choice of a user, the toolkit is distributed with a number of pre-fabricated physics lists for the convenience of many user applications. These physics lists are supported by the GEANT4 development team and can be recommended for specific physics tasks. However, based on the interests and needs of a specific project, a user may want to implement her or his own custom physics list.

The following sections offer several examples that show how to instantiate or select one or another pre-fabricated Physics List from the GEANT4 standard collection, as well as guidance composing a custom Physics List from pre-fabricated components or even entirely from scratch.

To view the contents of a Physics List, there are two useful methods: `DumpList()` and `DumpCutValueTable(G4int flag)`.

#### Reference Physics Lists

Number of ready to use Physics Lists are available with GEANT4 kernel. Below an example of instantiation of FTFP_BERT Physics List class is shown. The full set of reference Physics Lists is described in GEANT4 web.

Listing 6.2: Creating FTFP_BERT Physics List.

```
G4int verbose = 1;
FTFP_BERT* physlist = new FTFP_BERT(verbose);
runManager->SetUserInitialization(physlist);
```

### Building Physics List Using Factory

GEANT4 provides a class `G4PhysListFactory` allowing to defined Physics List by its name. The last for characters in the name defines an electromagnetic (EM) physics options. By default standard EM physics is used, "_EMV" corresponding to standard option1, "_EMX" - to standard option2, "_LIV" to EM Livermore physics, "_PEN" - to EM Penelope physics.

Listing 6.3: Creating Physics List by name.

```
G4int verbose = 1;
G4PhysListFactory factory;
G4VModularPhysicsList* physlist = factory.GetReferencePhysList("FTFP_BERT_EMV");
physlist.SetVerboseLevel(verbose);
runManager->SetUserInitialization(physlist);
```

The class `G4PhysListFactory` provides also another interface allowing to defined Physics List by the environment variable *PHYSLIST*.

Listing 6.4: Creating Physics List by name.

```
G4int verbose = 1;
G4PhysListFactory factory;
G4VModularPhysicsList* physlist = factory.ReferencePhysList();
physlist.SetVerboseLevel(verbose);
runManager->SetUserInitialization(physlist);
```

### Building Physics List from Physics Builders

Technically speaking, one can implement physics list in a "flat-out" manner, i.e. specify all necessary particles and associated processes in a single piece of code, as it will be shown later in this document. However, for practical purposes it is often more convenient to group together certain categories and make implementation more modular.

One very useful concept is a Modular Physics List, `G4VModularPhysicsList`, that is a sub-class of `G4VUserPhysicsLists` and allows a user to organize physics processes into "building blocks", or "modules", then compose a physics list of such modules. The concept allows to group together, at a relatively high level, desired combinations of selected particles and related processes. One of the advantages of such approach is that it allows to combine pre-fabricated physics modules that are centrally provided by GEANT4 kernel with user's applications.

`G4ModularPhysicsList` has all the functionalities as `G4VUserPhysicsList` class, plus several additional functionalities. One of the important methods is `RegisterPhysics(G4VPhysicsConstructor* )` for "registering" the above mentioned pre-fabricated physics modules. There also methods for removing or replacing physics modules.

Example below shows how `G4VModularPhysList` can be implemented.

Listing 6.5: Creating Physics List by name.

```
MyPhysicsList::MyPhysicsList():G4VModularPhysicsList()
{
  G4DataQuestionaire it(photon, neutron, no, no, no, neutronxs);
```

(continues on next page)

(continued from previous page)

```
  G4cout << "<<< Geant4 Physics List: MyPhysicsList " <<G4endl;
  G4cout <<G4endl;
  defaultCutValue = 0.7*mm;
  G4int ver = 1;
  SetVerboseLevel(ver);

  // EM Physics
  RegisterPhysics( new G4EmStandardPhysics(ver) );

  // Synchroton Radiation & GN Physics
  RegisterPhysics( new G4EmExtraPhysics(ver) );
  // Decays
  RegisterPhysics( new G4DecayPhysics(ver) );

  // Hadron physics
  RegisterPhysics( new G4HadronElasticPhysicsXS(ver) );
  RegisterPhysics( new G4QStoppingPhysics(ver) );
  RegisterPhysics( new G4IonBinaryCascadePhysics(ver) );
  RegisterPhysics( new G4HadronInelasticQBBC(ver));

  // Neutron tracking cut
  RegisterPhysics( new G4NeutronTrackingCut(ver) );
}
```

Note that each module to be registered with a Modular Physics List is a `G4VPhysicsConstructor` (or a derived object), i.e. a "sublist" that holds groups of particles and accompanying physics processes. A user can find these and other similar modules in the source/physics_lists/list area of GEANT4 core code, and can combine selected ones with custom modules, if desired.

In order to compose a custom physics module, two mandatory methods of a `G4VPhysicsConstructor` must be implemented: `ConstructParticle()` and `ConstructProcess()`; beyond that the implementation can be structured according to the developer's taste.

Another useful concept in the modular approach to composing a Physics List is the concept of so called "builders". This concept allows to encapsulate certain implementation details into smaller-scale software components, and offers the flexibility of re-using those component in different modules. At the general level, the scheme is this:

- Particles (hadrons) are created, and physics models to be used to simulate applicable processes are specified, usually in a particular range of validity.
- Physics processes for each particle type in the builder are created, and each process is outfitted with one or more hadronic physics models, as specified.
- If necessary, a cross section data set for a given particle type is added.

This concept is widely used through the GEANT4 hadronic domain, but the idea would be equally applicable in the electromagnetic area.

All builders can be found in the source/physics_lists/builders directory. There are basically two types of builders:

- Particle Builders
- Particle-Model Builders

A particle builder is somewhat "superior" here, as it specifies a particle or a group of particles, what category of processes are applicable, how to outfit a process with specified model(s), and how processes are to be registered with the `G4ProcessManager`. A particle-model builder instantiates a given model and implements details of associating it with one or more processes applicable to a given particle type. Some models can not be instantiated through a single interface class, but instead they need, in turn, to be composed from several components (examples are QGS and FTF).

Useful example builders to review and to consider as inspirations can be the following:

- G4PiKBuilder (.hh and .cc) - groups pions and kaons, together with a list of associated hadronic processes.
- G4BertiniPiKBuilder (.hh and .cc) - instantiates Bertini cascade model and implements how to outfit pion and kaon physics processes with this model. It also sets default validity range for the model.

- G4FTFPPiKBuilder (.hh and .cc) - composes a high energy FTF-based model and implements how to outfit hadronic processes for pions and kaons with the model. This example illustrates that a hadronic model does not always have a single interface class, but it needs to be created from several components. In particular, in this builder a "high energy generator" object (G4TheoFSGenerator) is created and is outfitted with G4FTFModel string model (which also gives this builder its name), we well as string fragmentation algorithm and intra-nuclear transport model. Please note that the quasi-elastic scattering is not set as FTF model has its own mechanism for it. A cross-section data set is specified for pions. A default validity range is also specified.

One detail to remember is that, in principle, the validity range for a given model can be setup for each particle type individually. But in these referenced applications the validity range is setup to be the same for a group of particles (i.e. for a number of corresponding inelastic hadronic processes). Once a builder is instantiated, one can override the default validity range (via SetMinEnergy or SetMaxEnergy methods), but the new value will be, again, given to a group of particles/processes. Also note that the validity range can be overridden only before calling the Build() method of a builder. Again, the approach is just a specifics of this particular implementation. Obviously, if a limited validity range is selected for a specific particle/model/process, one has to supplement another model or several models, to cover the entire range.

One more useful class is the `` G4PhysicsListHelper`` which is a service class that wraps around the technicalities of the physics process registering in GEANT4 and allows a user to easily associate a process with a particles, without knowing many details about various types of processes (discrete, continuous, etc.) and their internal ordering with `G4ProcessManager`. Curious users may eventually want to go deeper into details of G4ProcessManager class and, in particular, its group of `AddProcess(...)` methods, as it is the basis of G4PhysicsListHelper implementation. But for practical purposes, the use of G4PhysicsListHelper is likely to be sufficient in most cases.

Other useful details, including several elements of the software design philosophy and class diagrams, are given in *How to Specify Physics Processes*.

#### Building Physics List from Scratch

The user must derive a concrete class from `G4VUserPhysicsList` and implement three virtual methods:

- `ConstructParticle()` to instantiate each requested particle type;
- `ConstructPhysics()` to instantiate the desired physics processes and register each of them;
- `SetCuts(G4double aValue)` to set a cut value in range for all particles in the particle table, which invokes the rebuilding of the physics table.

At early stage of the initialisation of GEANT4 the method `ConstructParticle()` of `G4VUserPhysicsList` is invoked. The `ConstructProcess()` method must always invoke the `AddTransportation()` method in order to insure particle transportation. `AddTransportation()` must never be overridden. This is done automatically if `G4VUserPhysicsList` inherits of `G4VModularPhysicsList`. It is recommended for users as the most robust interface to Physics List. GEANT4 examples demonstrate different methods how to create user Physics List.

### 6.2.3 User Action Initialization

All user action classes must be defined through the protected method `SetUserAction()`. `Build()` methods should be used for defining user action classes for worker threads as well as for the sequential mode. `BuildForMaster()` should be used only for defining UserRunAction for the master thread. `BuildForMaster()` is not invoked in the sequential mode. In case the user uses his/her own `SteppingVerbose` class, it must be instantiated in the method `InitializeSteppingVerbose()` and returned.

### G4VUserActionInitialization

Listing 6.6: `G4VUserActionInitialization`

```
class G4VUserActionInitialization
{
  public:
    G4VUserActionInitialization();
    virtual ~G4VUserActionInitialization();

  public:
    virtual void Build() const = 0;
    virtual void BuildForMaster() const;
    virtual G4VSteppingVerbose* InitializeSteppingVerbose() const;

  protected:
    void SetUserAction(G4VUserPrimaryGeneratorAction*) const;
    void SetUserAction(G4UserRunAction*) const;
    void SetUserAction(G4UserEventAction*) const;
    void SetUserAction(G4UserStackingAction*) const;
    void SetUserAction(G4UserTrackingAction*) const;
    void SetUserAction(G4UserSteppingAction*) const;
};
```

### G4VUserPrimaryGeneratorAction

Listing 6.7: `G4VUserPrimaryGeneratorAction`

```
class G4VUserPrimaryGeneratorAction
{
  public:
    G4VUserPrimaryGeneratorAction();
    virtual ~G4VUserPrimaryGeneratorAction();

  public:
    virtual void GeneratePrimaries(G4Event* anEvent) = 0;
};
```

## 6.3 Optional User Actions

There are five virtual classes whose methods the user may override in order to gain control of the simulation at various stages. Each method of each action class has an empty default implementation, allowing the user to inherit and implement desired classes and methods.

Objects of user action classes must be registered with `G4RunManager` (*Manage the run procedures*), which takes ownership of them. The user must not delete these objects directly, and they must be created using ‘new’.

### 6.3.1 Usage of User Actions

#### `G4UserRunAction`

This class has three virtual methods which are invoked by `G4RunManager` for each run:

- **`GenerateRun()`** This method is invoked at the beginning of `BeamOn`. Because the user can inherit the class `G4Run` and create his/her own concrete class to store some information about the run, the `GenerateRun()` method is the place to instantiate such an object. It is also the ideal place to set variables which affect the physics table (such as production thresholds) for a particular run, because `GenerateRun()` is invoked before the calculation of the physics table.
- **`BeginOfRunAction()`** This method is invoked before entering the event loop. A typical use of this method would be to initialize and/or book histograms for a particular run. This method is invoked after the calculation of the physics tables.
- **`EndOfRunAction()`** This method is invoked at the very end of the run processing. It is typically used for a simple analysis of the processed run.

Listing 6.8: `G4UserRunAction`

```
class G4UserRunAction
{
  public:
    G4UserRunAction();
    virtual ~G4UserRunAction();

  public:
    virtual G4Run* GenerateRun();
    virtual void BeginOfRunAction(const G4Run*);
    virtual void EndOfRunAction(const G4Run*);
};
```

#### `G4UserEventAction`

This class has two virtual methods which are invoked by `G4EventManager` for each event:

- **`beginOfEventAction()`** This method is invoked before converting the primary particles to `G4Track` objects. A typical use of this method would be to initialize and/or book histograms for a particular event.
- **`endOfEventAction()`** This method is invoked at the very end of event processing. It is typically used for a simple analysis of the processed event. If the user wants to keep the currently processing event until the end of the current run, the user can invoke `fpEventManager->KeepTheCurrentEvent();` so that it is kept in `G4Run` object. This should be quite useful if you simulate quite many events and want to visualize only the most interest ones after the long execution. Given the memory size of an event and its contents may be large, it is the user's responsibility not to keep unnecessary events.

Listing 6.9: `G4UserEventAction`

```
class G4UserEventAction
{
  public:
      G4UserEventAction() {;}
      virtual ~G4UserEventAction() {;}
      virtual void BeginOfEventAction(const G4Event*);
      virtual void EndOfEventAction(const G4Event*);
  protected:
      G4EventManager* fpEventManager;
};
```

### G4UserStackingAction

This class has three virtual methods, `ClassifyNewTrack`, `NewStage` and `PrepareNewEvent` which the user may override in order to control the various track stacking mechanisms. ExampleN04 could be a good example to understand the usage of this class.

`ClassifyNewTrack()` is invoked by `G4StackManager` whenever a new `G4Track` object is "pushed" onto a stack by `G4EventManager`. `ClassifyNewTrack()` returns an enumerator, `G4ClassificationOfNewTrack`, whose value indicates to which stack, if any, the track will be sent. This value should be determined by the user. `G4ClassificationOfNewTrack` has four possible values:

- `fUrgent` - track is placed in the *urgent* stack
- `fWaiting` - track is placed in the *waiting* stack, and will not be simulated until the *urgent* stack is empty
- `fPostpone` - track is postponed to the next event
- `fKill` - the track is deleted immediately and not stored in any stack.

These assignments may be made based on the origin of the track which is obtained as follows:

```
G4int parent_ID = aTrack->get_parentID();
```

where

- `parent_ID = 0` indicates a primary particle
- `parent_ID > 0` indicates a secondary particle
- `parent_ID < 0` indicates postponed particle from previous event.

`NewStage()` is invoked when the *urgent* stack is empty and the *waiting* stack contains at least one `G4Track` object. Here the user may kill or re-assign to different stacks all the tracks in the *waiting* stack by calling the `stackManager->ReClassify()` method which, in turn, calls the `ClassifyNewTrack()` method. If no user action is taken, all tracks in the *waiting* stack are transferred to the *urgent* stack. The user may also decide to abort the current event even though some tracks may remain in the *waiting* stack by calling `stackManager->clear()`. This method is valid and safe only if it is called from the `G4UserStackingAction` class. A global method of event abortion is

```
G4UImanager * UImanager = G4UImanager::GetUIpointer();
UImanager->ApplyCommand("/event/abort");
```

`PrepareNewEvent()` is invoked at the beginning of each event. At this point no primary particles have been converted to tracks, so the *urgent* and *waiting* stacks are empty. However, there may be tracks in the *postponed-to-next-event* stack; for each of these the `ClassifyNewTrack()` method is called and the track is assigned to the appropriate stack.

Listing 6.10: `G4UserStackingAction`

```
#include "G4ClassificationOfNewTrack.hh"

class G4UserStackingAction
{
  public:
      G4UserStackingAction();
      virtual ~G4UserStackingAction();
  protected:
      G4StackManager * stackManager;

  public:
//---------------------------------------------------------------
// virtual methods to be implemented by user
//---------------------------------------------------------------
//
      virtual G4ClassificationOfNewTrack ClassifyNewTrack(const G4Track*);
```

(continues on next page)

(continued from previous page)

```
      virtual void NewStage();
      virtual void PrepareNewEvent();
};
```

### G4UserTrackingAction

Listing 6.11: G4UserTrackingAction

```
//---------------------------------------------------------------
// G4UserTrackingAction.hh
//
// Description:
// This class represents actions taken place by the user at
// the start/end point of processing one track.
//---------------------------------------------------------------

class G4UserTrackingAction
{
   public:

     // Constructor & Destructor
     G4UserTrackingAction(){};
     virtual ~G4UserTrackingAction(){}

     // Member functions
     virtual void PreUserTrackingAction(const G4Track*){}
     virtual void PostUserTrackingAction(const G4Track*){}

   protected:

     G4TrackingManager* fpTrackingManager;
};
```

### G4UserSteppingAction

Listing 6.12: G4UserSteppingAction

```
//---------------------------------------------------------------
//  G4UserSteppingAction.hh
//
//  Description:
//    This class represents actions taken place by the user at each
//    end of stepping.
//---------------------------------------------------------------

class G4UserSteppingAction
{
   public:

     // Constructor and destructor
     G4UserSteppingAction(){}
     virtual ~G4UserSteppingAction(){}

     // Member functions
     virtual void UserSteppingAction(const G4Step*){}

   protected:

     G4SteppingManager* fpSteppingManager;
```

(continues on next page)

(continued from previous page)

```
};
```

### 6.3.2 Killing Tracks in User Actions and Energy Conservation

In either of user action classes described in the previous section, the user can implement an unnatural/unphysical action. A typical example is to kill a track, which is under the simulation, in the user stepping action. In this case the user have to be cautious of the total energy conservation. The user stepping action itself does not take care the energy or any physics quantity associated with the killed track. Therefore if the user want to keep the total energy of an event in this case, the lost track energy need to be recorded by the user.

The same is true for user stacking or tracking actions. If the user has killed a track in these actions the all physics information associated with it would be lost and, for example, the total energy conservation be broken.

If the user wants the GEANT4 kernel to take care the total energy conservation automatically when he/she has killed artificially a track, the user has to use a killer process. For example if the user uses G4UserLimits and G4UserSpecialCuts process, energy of the killed track is added to the total energy deposit.

## 6.4 User Information Classes

Additional user information can be associated with various GEANT4 classes. There are basically two ways for the user to do this:

- derive concrete classes from base classes used in GEANT4. These are classes for run, hit, digit, trajectory and trajectory point, which are discussed in *Optional User Actions* for G4Run, *Hits* for G4VHit, *Digitization* for G4VDigit, and *Tracking* for G4VTrajectory and G4VTrajectoryPoint
- create concrete classes from provided abstract base classes and associate them with classes used in GEANT4. GEANT4 classes which can accommodate user information classes are G4Event, G4Track, G4PrimaryVertex, G4PrimaryParticle and G4Region. These classes are discussed here.

### 6.4.1 G4VUserEventInformation

`G4VUserEventInformation` is an abstract class from which the user can derive his/her own concrete class for storing user information associated with a G4Event class object. It is the user's responsibility to construct a concrete class object and set the pointer to a proper G4Event object.

Within a concrete implementation of G4UserEventAction, the SetUserEventInformation() method of G4EventManager may be used to set a pointer of a concrete class object to G4Event, given that the G4Event object is available only by "pointer to const". Alternatively, the user may modify the GenerateEvent() method of his/her own RunManager to instantiate a G4VUserEventInformation object and set it to G4Event.

The concrete class object is deleted by the GEANT4 kernel when the associated G4Event object is deleted.

### 6.4.2 G4VUserTrackInformation

This is an abstract class from which the user can derive his/her own concrete class for storing user information associated with a G4Track class object. It is the user's responsibility to construct a concrete class object and set the pointer to the proper G4Track object.

Within a concrete implementation of G4UserTrackingAction, the SetUserTrackInformation() method of G4TrackingManager may be used to set a pointer of a concrete class object to G4Track, given that the G4Track object is available only by "pointer to const".

The ideal place to copy a G4VUserTrackInformation object from a mother track to its daughter tracks is `G4UserTrackingAction::PostUserTrackingAction()`.

Listing 6.13: Copying `G4VUserTrackInformation` from mother to daughter tracks

```
void RE01TrackingAction::PostUserTrackingAction(const G4Track* aTrack)
{
  G4TrackVector* secondaries = fpTrackingManager->GimmeSecondaries();
  if(secondaries)
  {
    RE01TrackInformation* info = (RE01TrackInformation*)(aTrack->GetUserInformation());
    size_t nSeco = secondaries->size();
    if(nSeco>0)
    {
      for(size_t i=0; i < nSeco; i++)
      {
        RE01TrackInformation* infoNew = new RE01TrackInformation(info);
        (*secondaries)[i]->SetUserInformation(infoNew);
      }
    }
  }
}
```

The concrete class object is deleted by the GEANT4 kernel when the associated G4Track object is deleted. In case the user wants to keep the information, it should be copied to a trajectory corresponding to the track.

### 6.4.3 G4VUserPrimaryVertexInformation and G4VUserPrimaryTrackInformation

These abstract classes allow the user to attach information regarding the generated primary vertex and primary particle. Concrete class objects derived from these classes should be attached to `G4PrimaryVertex` and `G4PrimaryParticle` class objects, respectively.

The concrete class objects are deleted by the GEANT4 kernel when the associated G4PrimaryVertex or G4PrimaryParticle class objects are deleted along with the deletion of G4Event.

### 6.4.4 G4VUserRegionInformation

This abstract base class allows the user to attach information associated with a region. For example, it would be quite beneficial to add some methods returning a Boolean flag to indicate the characteristics of the region (e.g. tracker, calorimeter, etc.). With this example, the user can easily and quickly identify the detector component.

Listing 6.14: A sample region information class

```
class RE01RegionInformation : public G4VUserRegionInformation
{
  public:
    RE01RegionInformation();
```

(continues on next page)

(continued from previous page)

```
    ~RE01RegionInformation();
    void Print() const;

  private:
    G4bool isWorld;
    G4bool isTracker;
    G4bool isCalorimeter;

  public:
    inline void SetWorld(G4bool v=true) {isWorld = v;}
    inline void SetTracker(G4bool v=true) {isTracker = v;}
    inline void SetCalorimeter(G4bool v=true) {isCalorimeter = v;}
    inline G4bool IsWorld() const {return isWorld;}
    inline G4bool IsTracker() const {return isTracker;}
    inline G4bool IsCalorimeter() const {return isCalorimeter;}
};
```

The following code is an example of a stepping action. Here, a track is suspended when it enters the "calorimeter region" from the "tracker region".

Listing 6.15: Sample use of a region information class

```
void RE01SteppingAction::UserSteppingAction(const G4Step * theStep)
{
  // Suspend a track if it is entering into the calorimeter

  // check if it is alive
  G4Track * theTrack = theStep->GetTrack();
  if(theTrack->GetTrackStatus()!=fAlive) { return; }

  // get region information
  G4StepPoint * thePrePoint = theStep->GetPreStepPoint();
  G4LogicalVolume * thePreLV = thePrePoint->GetPhysicalVolume()->GetLogicalVolume();
  RE01RegionInformation* thePreRInfo
   = (RE01RegionInformation*)(thePreLV->GetRegion()->GetUserInformation());
  G4StepPoint * thePostPoint = theStep->GetPostStepPoint();
  G4LogicalVolume * thePostLV = thePostPoint->GetPhysicalVolume()->GetLogicalVolume();
  RE01RegionInformation* thePostRInfo
   = (RE01RegionInformation*)(thePostLV->GetRegion()->GetUserInformation());

  // check if it is entering to the calorimeter volume
  if(!(thePreRInfo->IsCalorimeter()) && (thePostRInfo->IsCalorimeter()))
  { theTrack->SetTrackStatus(fSuspend); }
}
```

## 6.5 Multiple User Actions

Starting from GEANT4 Version 10.3 it is possible to attach multiple instances of the same type of user action to a single run manager. This is achieved via the use of a special proxy classes to which multiple child user actions are attached. This is allowed for run-, event-, tracking- and stepping-type user actions (`G4UserRunAction`, `G4UserEventAction`, `G4UserTrackingAction`, `G4UserSteppingAction`).

The kernel still sees a single user action of each type, the proxy will forward the calls from kernel to all the attached child user actions.

Listing 6.16: An example of the use of the use of multiple user-actions.

```
#include "G4MultiRunAction.hh"
#include "G4MultiEventAction.hh"
```

(continues on next page)

(continued from previous page)

```
#include "G4MultiTrackingAction.hh"
#include "G4MultiSteppingAction.hh"
//...
void MyUserActionInitialization::Build()
{
  //...
  // Example with multiple-event action, similarly
  // for the other cases
  // multi- user actions extend std::vector
  auto multiAction = new G4MultiEventAction { new MyEventAction1, new MyEventAction2 } ;
  //...
  multiAction->push_back( new MyEventAction3 );
  SetUserAction( multiAction );
  //...
}
```

### 6.5.1 Exceptions

This functionality is not implemented for the the stacking user action and primary generation action. There is no multiple `G4UserStackingAction` equivalent since this would require a complex handling of the case in which conflicting classifications are issued. For the case of `G4VUserPrimaryGeneratorAction` the use case of the multiple user actions is already addressed by the design of the class itself. User can implement one or more generators in the actions.

For the case of `G4MultiRunAction` only one of the child user actions can implement the `G4UserRunAction::GenerateRun()` method returning a non null, user derived `G4Run` object, otherwise an exception is thrown.

# CHAPTER SEVEN

# CONTROL

## 7.1 Built-in Commands

GEANT4 has various built-in user interface commands, each of which corresponds roughly to a GEANT4 category. These commands can be used

- interactively via a (Graphical) User Interface - (G)UI,
- in a macro file via /control/execute <command>,
- within C++ code with the ApplyCommand method of G4UImanager.

**Note:** The availability of individual commands, the ranges of parameters, the available candidates on individual command parameters vary according to the implementation of your application and may even vary dynamically during the execution of your job.

The following is a short summary of available commands. You can also see the all available commands by executing 'help' in your UI session.

- List of built-in commands

## 7.2 User Interface - Defining New Commands

### 7.2.1 G4UImessenger

`G4UImessenger` is a base class which represents a messenger that delivers command(s) to the destination class object. Concrete messengers are instantiated by, and owned by, the functional classes for which they provide a user interface; messengers should be deleted by those classes in their own destructors.

Your concrete messenger should have the following functionalities.

- Construct your command(s) in the constructor of your messenger.
- Destruct your command(s) in the destructor of your messenger.

These requirements mean that your messenger should keep all pointers to your command objects as its data members.

You can use `G4UIcommand` derived classes for the most frequent types of command. These derived classes have their own conversion methods according to their types, and they make implementation of the `SetNewValue()` and `GetCurrentValue()` methods of your messenger much easier and simpler.

`G4UIcommand` objects are owned by the messenger. If instantiated via *new*, they should be deleted in the messenger destructor.

For complicated commands which take various parameters, you can use the `G4UIcommand` base class, and construct `G4UIparameter` objects by yourself. You don't need to delete `G4UIparameter` object(s).

In the `SetNewValue()` and `GetCurrentValue()` methods of your messenger, you can compare the `G4UIcommand` pointer given in the argument of these methods with the pointer of your command, because your messenger keeps the pointers to the commands. Thus, you don't need to compare by command name. Please remember, in the cases where you use `G4UIcommand` derived classes, you should store the pointers with the types of these derived classes so that you can use methods defined in the derived classes according to their types without casting.

`G4UImanager/G4UIcommand/G4UIparameter` have very powerful type and range checking routines. You are strongly recommended to set the range of your parameters. For the case of a numerical value (`int` or `double`), the range can be given by a `G4String` using C++ notation, e.g., `"X > 0 && X < 10"`. For the case of a string type parameter, you can set a candidate list. Please refer to the detailed descriptions below.

`GetCurrentValue()` will be invoked after the user's application of the corresponding command, and before the `SetNewValue()` invocation. This `GetCurrentValue()` method will be invoked only if

- at least one parameter of the command has a range
- at least one parameter of the command has a candidate list
- at least the value of one parameter is omitted and this parameter is defined as omittable and currentValueAsDefault

For the first two cases, you can re-set the range or the candidate list if you need to do so, but these "re-set" parameters are needed only for the case where the range or the candidate list varies dynamically.

A command can be "state sensitive", i.e., the command can be accepted only for a certain `G4ApplicationState`(s). For example, the `/run/beamOn` command should not be accepted when GEANT4 is processing another event ("G4State_EventProc" state). You can set the states available for the command with the `AvailableForStates()` method.

### 7.2.2 G4UIcommand and its derived classes

#### Methods available for all derived classes

These are methods defined in the `G4UIcommand` base class which should be used from the derived classes.

- `void SetGuidance(char*)`
  Define a guidance line. You can invoke this method as many times as you need to give enough amount of guidance. Please note that the first line will be used as a title head of the command guidance.
- `void availableForStates(G4ApplicationState s1,...)`
  If your command is valid only for certain states of the GEANT4 kernel, specify these states by this method. Currently available states are `G4State_PreInit`, `G4State_Init`, `G4State_Idle`, `G4State_GeomClosed`, and `G4State_EventProc`. Refer to the *as a state machine* for meaning of each state. Please note that the `Pause` state had been removed from `G4ApplicationState`.
- `void SetRange(char* range)`
  Define a range of the parameter(s). Use C++ notation, e.g., `"x > 0 && x < 10"`, with variable name(s) defined by the `SetParameterName()` method. For the case of a `G4ThreeVector`, you can set the relation between parameters, e.g., `"x > y"`.

### G4UIdirectory

This is a `G4UIcommand` derived class for defining a directory containing commands. It is owned by, and should be deleted in the destructor of, the associated G4UImessenger class, after all of its contained commands have been deleted.

- `G4UIdirectory(char* directoryPath)`
  Constructor. Argument is the (full-path) directory, which must begin and terminate with "`/`:.

### G4UIcmdWithoutParameter

This is a `G4UIcommand` derived class for a command which takes no parameter.

- `G4UIcmdWithoutParameter(char* commandPath, G4UImessenger* theMessenger)`
  Constructor. Arguments are the (full-path) command name and the pointer to your messenger.

### G4UIcmdWithABool

This is a `G4UIcommand` derived class which takes one Boolean type parameter.

- `G4UIcmdWithABool(char* commandpath,G4UImanager* theMessenger)`
  Constructor. Arguments are the (full-path) command name and the pointer to your messenger.
- `void SetParameterName(char* paramName, G4bool omittable)`
  Define the name of the Boolean parameter and set the omittable flag. If omittable is true, you should define the default value using the next method.
- `void SetDefaultValue(G4bool defVal)`
  Define the default value of the Boolean parameter.
- `G4bool GetNewBoolValue(G4String paramString)`
  Convert `G4String` parameter value given by the `SetNewValue()` method of your messenger into Boolean.
- `G4String convertToString(G4bool currVal)`
  Convert the current Boolean value to `G4String` which should be returned by the `GetCurrentValue()` method of your messenger.

### G4UIcmdWithAnInteger

This is a `G4UIcommand` derived class which takes one integer type parameter.

- `G4UIcmdWithAnInteger(char* commandpath, G4UImanager* theMessenger)`
  Constructor. Arguments are the (full-path) command name and the pointer to your messenger.
- `void SetParameterName(char* paramName, G4bool omittable)`
  Define the name of the integer parameter and set the omittable flag. If omittable is true, you should define the default value using the next method.
- `void SetDefaultValue(G4int defVal)`
  Define the default value of the integer parameter.
- `G4int GetNewIntValue(G4String paramString)`
  Convert `G4String` parameter value given by the `SetNewValue()` method of your messenger into integer.
- `G4String convertToString(G4int currVal)`
  Convert the current integer value to `G4String`, which should be returned by the `GetCurrentValue()` method of your messenger.

### G4UIcmdWithADouble

This is a `G4UIcommand` derived class which takes one double type parameter.

- `G4UIcmdWithADouble(char* commandpath, G4UImanager* theMessenger)`
  Constructor. Arguments are the (full-path) command name and the pointer to your messenger.
- `void SetParameterName(char* paramName, G4bool omittable)`
  Define the name of the double parameter and set the omittable flag. If omittable is true, you should define the default value using the next method.
- `void SetDefaultValue(G4double defVal)`
  Define the default value of the double parameter.
- `G4double GetNewDoubleValue(G4String paramString)`
  Convert `G4String` parameter value given by the `SetNewValue()` method of your messenger into double.
- `G4String convertToString(G4double currVal)`
  Convert the current double value to `G4String` which should be returned by the `GetCurrentValue()` method of your messenger.

### G4UIcmdWithAString

This is a `G4UIcommand` derived class which takes one string type parameter.

- `G4UIcmdWithAString(char* commandpath, G4UImanager* theMessenger)`
  Constructor. Arguments are the (full-path) command name and the pointer to your messenger.
- `void SetParameterName(char* paramName, G4bool omittable)`
  Define the name of the string parameter and set the omittable flag. If omittable is true, you should define the default value using the next method.
- `void SetDefaultValue(char* defVal)`
  Define the default value of the string parameter.
- `void SetCandidates(char* candidateList)`
  Define a candidate list which can be taken by the parameter. Each candidate listed in this list should be separated by a single space. If this candidate list is given, a string given by the user but which is not listed in this list will be rejected.

### G4UIcmdWith3Vector

This is a `G4UIcommand` derived class which takes one three vector parameter.

- `G4UIcmdWith3Vector(char* commandpath, G4UImanager* theMessenger)`
  Constructor. Arguments are the (full-path) command name and the pointer to your messenger.
- `void SetParameterName(char* paramNamX, char* paramNamY, char* paramNamZ, G4bool omittable)`
  Define the names of each component of the three vector and set the omittable flag. If omittable is true, you should define the default value using the next method.
- `void SetDefaultValue(G4ThreeVector defVal)`
  Define the default value of the three vector.
- `G4ThreeVector GetNew3VectorValue(G4String paramString)`
  Convert the `G4String` parameter value given by the `SetNewValue()` method of your messenger into a `G4ThreeVector`.
- `G4String convertToString(G4ThreeVector currVal)`
  Convert the current three vector to `G4String`, which should be returned by the `GetCurrentValue()` method of your messenger.

### G4UIcmdWithADoubleAndUnit

This is a `G4UIcommand` derived class which takes one double type parameter and its unit.

- `G4UIcmdWithADoubleAndUnit(char* commandpath, G4UImanager* theMessenger)`
  Constructor. Arguments are the (full-path) command name and the pointer to your messenger.
- `void SetParameterName(char* paramName, G4bool omittable)`
  Define the name of the double parameter and set the omittable flag. If omittable is true, you should define the default value using the next method.
- `void SetDefaultValue(G4double defVal)`
  Define the default value of the double parameter.
- `void SetUnitCategory(char* unitCategory)`
  Define acceptable unit category.
- `void SetDefaultUnit(char* defUnit)`
  Define the default unit. Please use this method and the `SetUnitCategory()` method alternatively.
- `G4double GetNewDoubleValue(G4String paramString)`
  Convert `G4String` parameter value given by the `SetNewValue()` method of your messenger into double. Please note that the return value has already been multiplied by the value of the given unit.
- `G4double GetNewDoubleRawValue(G4String paramString)`
  Convert `G4String` parameter value given by the `SetNewValue()` method of your messenger into double but without multiplying the value of the given unit.
- `G4double GetNewUnitValue(G4String paramString)`
  Convert `G4String` unit value given by the `SetNewValue()` method of your messenger into double.
- `G4String convertToString(G4bool currVal, char* unitName)`
  Convert the current double value to a `G4String`, which should be returned by the `GetCurrentValue()` method of your messenger. The double value will be divided by the value of the given unit and converted to a string. Given unit will be added to the string.

### G4UIcmdWith3VectorAndUnit

This is a `G4UIcommand` derived class which takes one three vector parameter and its unit.

- `G4UIcmdWith3VectorAndUnit(char* commandpath, G4UImanager* theMessenger)`
  Constructor. Arguments are the (full-path) command name and the pointer to your messenger.
- `void SetParameterName(char* paramNamX, char* paramNamY, char* paramNamZ, G4bool omittable)`
  Define the names of each component of the three vector and set the omittable flag. If omittable is true, you should define the default value using the next method.
- `void SetDefaultValue(G4ThreeVector defVal)`
  Define the default value of the three vector.
- `void SetUnitCategory(char* unitCategory)`
  Define acceptable unit category.
- `void SetDefaultUnit(char* defUnit)`
  Define the default unit. Please use this method and the `SetUnitCategory()` method alternatively.
- `G4ThreeVector GetNew3VectorValue(G4String paramString)`
  Convert a `G4String` parameter value given by the `SetNewValue()` method of your messenger into a `G4ThreeVector`. Please note that the return value has already been multiplied by the value of the given unit.
- `G4ThreeVector GetNew3VectorRawValue(G4String paramString)`
  Convert a `G4String` parameter value given by the `SetNewValue()` method of your messenger into three vector, but without multiplying the value of the given unit.
- `G4double GetNewUnitValue(G4String paramString)`
  Convert a `G4String` unit value given by the `SetNewValue()` method of your messenger into a double.
- `G4String convertToString(G4ThreeVector currVal, char* unitName)`

Convert the current three vector to a G4String which should be returned by the GetCurrentValue() method of your messenger. The three vector value will be divided by the value of the given unit and converted to a string. Given unit will be added to the string.

#### Additional comments on the SetParameterName() method

You can add one additional argument of G4bool type for every SetParameterName() method mentioned above. This additional argument is named currentAsDefaultFlag and the default value of this argument is false. If you assign this extra argument as true, the default value of the parameter will be overridden by the current value of the target class.

### 7.2.3 An example messenger

This example is of G4ParticleGunMessenger, which is made by inheriting G4UIcommand.

Listing 7.1: An example of G4ParticleGunMessenger.hh.

```
#ifndef G4ParticleGunMessenger_h
#define G4ParticleGunMessenger_h 1

class G4ParticleGun;
class G4ParticleTable;
class G4UIcommand;
class G4UIdirectory;
class G4UIcmdWithoutParameter;
class G4UIcmdWithAString;
class G4UIcmdWithADoubleAndUnit;
class G4UIcmdWith3Vector;
class G4UIcmdWith3VectorAndUnit;

#include "G4UImessenger.hh"
#include "globals.hh"

class G4ParticleGunMessenger: public G4UImessenger
{
  public:
    G4ParticleGunMessenger(G4ParticleGun * fPtclGun);
    ~G4ParticleGunMessenger();

  public:
    void SetNewValue(G4UIcommand * command,G4String newValues);
    G4String GetCurrentValue(G4UIcommand * command);

  private:
    G4ParticleGun * fParticleGun;
    G4ParticleTable * particleTable;

  private: //commands
    G4UIdirectory *             gunDirectory;
    G4UIcmdWithoutParameter *   listCmd;
    G4UIcmdWithAString *        particleCmd;
    G4UIcmdWith3Vector *        directionCmd;
    G4UIcmdWithADoubleAndUnit * energyCmd;
    G4UIcmdWith3VectorAndUnit * positionCmd;
    G4UIcmdWithADoubleAndUnit * timeCmd;

};

#endif
```

Listing 7.2: An example of `G4ParticleGunMessenger.cc`.

```
#include "G4ParticleGunMessenger.hh"
#include "G4ParticleGun.hh"
#include "G4Geantino.hh"
#include "G4ThreeVector.hh"
#include "G4ParticleTable.hh"
#include "G4UIdirectory.hh"
#include "G4UIcmdWithoutParameter.hh"
#include "G4UIcmdWithAString.hh"
#include "G4UIcmdWithADoubleAndUnit.hh"
#include "G4UIcmdWith3Vector.hh"
#include "G4UIcmdWith3VectorAndUnit.hh"
#include <iostream.h>

G4ParticleGunMessenger::G4ParticleGunMessenger(G4ParticleGun * fPtclGun)
:fParticleGun(fPtclGun)
{
  particleTable = G4ParticleTable::GetParticleTable();

  gunDirectory = new G4UIdirectory("/gun/");
  gunDirectory->SetGuidance("Particle Gun control commands.");

  listCmd = new G4UIcmdWithoutParameter("/gun/list",this);
  listCmd->SetGuidance("List available particles.");
  listCmd->SetGuidance(" Invoke G4ParticleTable.");

  particleCmd = new G4UIcmdWithAString("/gun/particle",this);
  particleCmd->SetGuidance("Set particle to be generated.");
  particleCmd->SetGuidance(" (geantino is default)");
  particleCmd->SetParameterName("particleName",true);
  particleCmd->SetDefaultValue("geantino");
  G4String candidateList;
  G4int nPtcl = particleTable->entries();
  for(G4int i=0;i<nPtcl;i++)
  {
    candidateList += particleTable->GetParticleName(i);
    candidateList += " ";
  }
  particleCmd->SetCandidates(candidateList);

  directionCmd = new G4UIcmdWith3Vector("/gun/direction",this);
  directionCmd->SetGuidance("Set momentum direction.");
  directionCmd->SetGuidance("Direction needs not to be a unit vector.");
  directionCmd->SetParameterName("Px","Py","Pz",true,true);
  directionCmd->SetRange("Px != 0 || Py != 0 || Pz != 0");

  energyCmd = new G4UIcmdWithADoubleAndUnit("/gun/energy",this);
  energyCmd->SetGuidance("Set kinetic energy.");
  energyCmd->SetParameterName("Energy",true,true);
  energyCmd->SetDefaultUnit("GeV");
  energyCmd->SetUnitCandidates("eV keV MeV GeV TeV");

  positionCmd = new G4UIcmdWith3VectorAndUnit("/gun/position",this);
  positionCmd->SetGuidance("Set starting position of the particle.");
  positionCmd->SetParameterName("X","Y","Z",true,true);
  positionCmd->SetDefaultUnit("cm");
  positionCmd->SetUnitCandidates("micron mm cm m km");

  timeCmd = new G4UIcmdWithADoubleAndUnit("/gun/time",this);
  timeCmd->SetGuidance("Set initial time of the particle.");
  timeCmd->SetParameterName("t0",true,true);
  timeCmd->SetDefaultUnit("ns");
  timeCmd->SetUnitCandidates("ns ms s");

  // Set initial value to G4ParticleGun
```

(continues on next page)

(continued from previous page)

```
  fParticleGun->SetParticleDefinition( G4Geantino::Geantino() );
  fParticleGun->SetParticleMomentumDirection( G4ThreeVector(1.0,0.0,0.0) );
  fParticleGun->SetParticleEnergy( 1.0*GeV );
  fParticleGun->SetParticlePosition(G4ThreeVector(0.0*cm, 0.0*cm, 0.0*cm));
  fParticleGun->SetParticleTime( 0.0*ns );
}
```

```
G4ParticleGunMessenger::~G4ParticleGunMessenger()
{
  delete listCmd;
  delete particleCmd;
  delete directionCmd;
  delete energyCmd;
  delete positionCmd;
  delete timeCmd;
  delete gunDirectory;
}

void G4ParticleGunMessenger::SetNewValue(
  G4UIcommand * command,G4String newValues)
{
  if( command==listCmd )
  { particleTable->dumpTable(); }
  else if( command==particleCmd )
  {
    G4ParticleDefinition* pd = particleTable->findParticle(newValues);
    if(pd != NULL)
    { fParticleGun->SetParticleDefinition( pd ); }
  }
  else if( command==directionCmd )
  { fParticleGun->SetParticleMomentumDirection(directionCmd->
     GetNew3VectorValue(newValues)); }
  else if( command==energyCmd )
  { fParticleGun->SetParticleEnergy(energyCmd->
     GetNewDoubleValue(newValues)); }
  else if( command==positionCmd )
  { fParticleGun->SetParticlePosition(
     directionCmd->GetNew3VectorValue(newValues)); }
  else if( command==timeCmd )
  { fParticleGun->SetParticleTime(timeCmd->
     GetNewDoubleValue(newValues)); }
}

G4String G4ParticleGunMessenger::GetCurrentValue(G4UIcommand * command)
{
  G4String cv;

  if( command==directionCmd )
  { cv = directionCmd->ConvertToString(
     fParticleGun->GetParticleMomentumDirection()); }
  else if( command==energyCmd )
  { cv = energyCmd->ConvertToString(
     fParticleGun->GetParticleEnergy(),"GeV"); }
  else if( command==positionCmd )
  { cv = positionCmd->ConvertToString(
     fParticleGun->GetParticlePosition(),"cm"); }
  else if( command==timeCmd )
  { cv = timeCmd->ConvertToString(
     fParticleGun->GetParticleTime(),"ns"); }
  else if( command==particleCmd )
  { // update candidate list
    G4String candidateList;
    G4int nPtcl = particleTable->entries();
    for(G4int i=0;i<nPtcl;i++)
    {
```

(continues on next page)

(continued from previous page)

```
      candidateList += particleTable->GetParticleName(i);
      candidateList += " ";
    }
    particleCmd->SetCandidates(candidateList);
  }
  return cv;
}
```

### 7.2.4 How to control the output of G4cout/G4cerr

Instead of *std::cout* and *std::cerr*, GEANT4 uses `G4cout` and `G4cerr`. Output streams from `G4cout/G4cerr` are handled by `G4UImanager` which allows the application programmer to control the flow of the stream. Output strings may therefore be displayed on another window or stored in a file. This is accomplished as follows:

1. Derive a class from `G4UIsession` and implement the two methods:

   ```
   G4int ReceiveG4cout(const G4String& coutString);
   G4int ReceiveG4cerr(const G4String& cerrString);
   ```

   These methods receive the string stream of `G4cout` and `G4cerr`, respectively. The string can be handled to meet specific requirements. The following sample code shows how to make a log file of the output stream:

   ```
   ostream logFile;
   logFile.open("MyLogFile");
   G4int MySession::ReceiveG4cout(const G4String& coutString)
   {
     logFile << coutString << flush;
     return 0;
   }
   ```

2. Set the destination of `G4cout/G4cerr` using `G4UImanager::SetCoutDestination(session)`. Typically this method is invoked from the constructor of `G4UIsession` and its derived classes, such as `G4UIGAG/G4UIteminal`. This method sets the destination of `G4cout/G4cerr` to the session. For example, when the following code appears in the constructor of `G4UIterminal`, the method `SetCoutDestination(this)` tells *UImanager* that this instance of `G4UIterminal` receives the stream generated by `G4cout`.

   ```
   G4UIterminal::G4UIterminal()
   {
     UI = G4UImanager::GetUIpointer();
     UI->SetCoutDestination(this);
     //  ...
   }
   ```

   Similarly, `UI->SetCoutDestination(NULL)` must be added to the destructor of the class.

3. Write or modify the main program. To modify `exampleB1` to produce a log file, derive a class as described in step 1 above, and add the following lines to the main program:

   ```
   #include "MySession.hh"
   main()
   {
     // get the pointer to the User Interface manager
     G4UImanager* UI = G4UImanager::GetUIpointer();
     // construct a session which receives G4cout/G4cerr
     MySession * LoggedSession = new MySession;
     UI->SetCoutDestination(LoggedSession);
     // session->SessionStart(); // not required in this case
     // .... do simulation here ...

     delete LoggedSession;
   ```

(continues on next page)

(continued from previous page)

```
  return 0;
}
```

**Note:** `G4cout/G4cerr` should not be used in the constructor of a class if the instance of the class is intended to be used as `static`. This restriction comes from the language specification of C++. See the documents below for details:

- M.A.Ellis, B.Stroustrup, “Annotated C++ Reference Manual”, Section 3.4 [Ellis1990]
- P.J.Plauger, “The Draft Standard C++ Library” [Plauger1995]

CHAPTER

EIGHT

# VISUALIZATION

## 8.1 Introduction to Visualization

The GEANT4 visualization system was developed in response to a diverse set of requirements:

1. Quick response to study geometries, trajectories and hits
2. High-quality output for publications
3. Flexible camera control to debug complex geometries
4. Tools to show volume overlap errors in detector geometries
5. Interactive picking to get more information on visualized objects

No one graphics system is ideal for all of these requirements, and many of the large software frameworks into which GEANT4 has been incorporated already have their own visualization systems, so GEANT4 visualization was designed around an abstract interface that supports a diverse family of graphics systems. Some of these graphics systems use a graphics library compiled with GEANT4, such as OpenGL, Qt, while others involve a separate application, such as HepRApp or DAWN.

Most examples include a vis.mac to perform typical visualization for that example. The macro includes optional code which you can uncomment to activate additional visualization features.

### 8.1.1 What Can be Visualized

Simulation data can be visualized:

- Detector components
  - A hierarchical structure of physical volumes
  - A piece of physical volume, logical volume, and solid
- Particle trajectories and tracking steps
- Hits of particles in detector components
- Scoring data
- Plots

Other user defined objects can be visualized:

- Polylines, such as coordinate axes
- 3D Markers, such as eye guides
- Text, descriptive character strings, comments or titles
- Scales
- Logos

### 8.1.2 You have a Choice of Visualization Drivers

The many graphics systems that GEANT4 supports are complementary to each other.

- OpenGL
  - View directly from GEANT4
  - Requires addition of GL libraries that are freely available for all operating systems (and pre-installed on many)
  - Rendered, photorealistic image with some interactive features
  - zoom, rotate, translate
  - Fast response (can usually exploit full potential of graphics hardware)
  - Print to EPS (vector and pixel graphics)
- Qt
  - View directly from GEANT4
  - Requires addition of Qt and GL libraries that are freely available on most operating systems
  - Rendered, photorealistic image
  - Many interactive features
  - zoom, rotate, translate
  - Fast response (can usually exploit full potential of graphics hardware)
  - Expanded printing ability (vector and pixel graphics)
  - Easy interface to make movies
- Open Inventor
  - View directly from GEANT4
  - Requires addition of Coin3d libraries (freely available for most Linux systems).
  - Rendered, photorealistic image
  - Many interactive features
  - zoom, rotate, translate
  - click to “see inside” opaque volumes
  - Fast response (can usually exploit full potential of graphics hardware)
  - Expanded printing ability: vector graphics (with transparency in PDF) and pixel graphics
- Qt3D
  - View directly from GEANT4
  - Requires addition of Qt libraries that are freely available on most operating systems
  - Rendered, photorealistic image
  - Many interactive features
  - zoom, rotate, translate
  - Fast response (can usually exploit full potential of graphics hardware)
- TSG
  - View directly from GEANT4
  - Based on the ToolsSG library distributed with Geant4.
  - Rendered, photorealistic image
  - Many interactive features
  - zoom, rotate, translate
  - Fast response (can usually exploit full potential of graphics hardware)
  - Can also view plots of histograms, etc., accumulated in your application
- Vtk
  - View directly from GEANT4
  - Requires installation of Vtk (vtk.org)
  - Rendered, photorealistic image
  - Many interactive features
  - zoom, rotate, translate
  - Fast response (can usually exploit full potential of graphics hardware)
- HepRepFile
  - Create a file to view in a HepRep browser such as HepRApp or FRED

  - Requires a HepRep browser (above options work on any operating system)
  - Wireframe or simple area fills (not photorealistic)
  - Many interactive features
  - zoom, rotate, translate
  - click to show attributes (momentum, etc.)
  - special projections (FishEye, etc.)
  - control visibility from hierarchical (tree) view of data
  - Hierarchical view of the geometry
  - Export to many vector graphic formats (PostScript, PDF, etc.)
- DAWN
  - Create a file to view in the DAWN Renderer
  - Requires DAWN, available for all Linux and Windows systems
  - Rendered, photorealistic image
  - No interactive features
  - Highest quality technical rendering - output to vector PostScript
- VRML
  - Create a file with VRML2FILE to view in any VRML browser (many different choices for different browsers and operating systems).
  - Rendered, photorealistic image with some interactive features
  - zoom, rotate, translate
  - Limited printing ability (pixel graphics, not vector graphics)
- RayTracer
  - Create a jpeg file
  - Forms image by using GEANT4's own tracking to follow photons through the detector
  - Can show geometry but not trajectories
  - Can render any geometry that GEANT4 can handle (such as Boolean solids)
  - Supports shadows, transparency and mirrored surfaces
- gMocren
  - Create a gMocren file suitable for viewing in the gMocren volume data visualization application
  - Represents three dimensional volume data such as radiation therapy dose
  - Can also include geometry and trajectory information
- ASCIITree
  - Text dump of the geometry hierarchy
  - Not graphical
  - Control over level of detail to be dumped
  - Can calculate mass and volume of any hierarchy of volumes

## 8.1.3 Choose the Driver that Meets Your Needs

- If you want very responsive photorealistic graphics (and have the OpenGL libraries installed)
  - OpenGL is a good solution (if you have the Motif extensions, this also gives GUI control)
- If you want to have the User Interface and all Visualization windows in the same window
  - Only Qt can do that
- If you want very responsive photorealistic graphics plus more interactivity (and have the OpenInventor or Qt libraries installed)
  - OpenInventor with or without Qt are good solutions
  - Qt3D, TSG and Vtk are also candidates
- If you want GUI control, very responsive photorealistic graphics plus more interactivity (and have the Qt libraries installed).
  - Qt is a good solution
- If you want GUI control, want to be able to pick on items to inquire about them (identity, momentum, etc.), perhaps want to render to vector formats, and a wireframe look will do
  - HepRepFile will meet your needs

- If you want to render highest quality photorealistic images for use in a poster or a technical design report, and you can live without quick rotate and zoom
  - DAWN is the way to go
- If you want to render to a 3D format that others can view in a variety of commodity browsers (including some web browser plug-ins)
  - VRML is the way to go
- If you want to visualize a geometry that the other visualization drivers can't handle, or you need transparency or mirrors, and you don't need to visualize trajectories
  - RayTracer will do it
- If you want to visualization volume data, such as radiation therapy dose distributions
  - gMocren will meet your needs
- If you just want to quickly check the geometry hierarchy, or if you want to calculate the volume or mass of any geometry hierarchy
  - ASCIITree will meet your needs
- You can also add your own visualization driver.
  - GEANT4's visualization system is modular. By creating just three new classes, you can direct GEANT4 information to your own visualization system.

### 8.1.4 Controlling Visualization

Your GEANT4 code stays basically the same no matter which driver you use.

Visualization is performed either with commands or from C++ code.

- Some visualization drivers work directly from GEANT4
  - OpenGL
  - Qt
  - Qt3D
  - Vtk
  - TSG
  - OpenInventor
  - RayTracer
  - ASCIITree
- For other visualization drivers, you first have GEANT4 produce a file, and then you have that file rendered by another application (which may have GUI control)
  - HepRepFile
  - DAWN
  - VRML2FILE
  - gMocren

### 8.1.5 Visualization Details

The following sections of this guide cover the details of GEANT4 visualization:

- Adding Visualization to Your Executable
- The Visualization Drivers
- Controlling Visualization from Commands
- Controlling Visualization from Compiled Code
- Visualization Attributes
- Enhanced Trajectory Drawing
- Polylines, Markers and Text
- Making a Movie

Other useful references for GEANT4 visualization outside of this user guide:

- Macro files `vis.mac` distributed in GEANT4 source in basic examples.

## 8.2 Adding Visualization to Your Executable

This section explains how to incorporate your selected visualization drivers into the `main()` function and create an executable for it. In order to perform visualization with your GEANT4 executable, you must compile it with support for the required visualization driver(s). You may be dazzled by the number of choices of visualization driver, but you need not use all of them at one time.

### 8.2.1 Installing Visualization Drivers

Depending on what has been installed on your system and how the Geant4 install you are using was configured, several kinds of visualization driver are available. One or many drivers may be chosen for realization in compilation, depending on your visualization requirements. Features and notes on each driver are briefly described in *The Visualization Drivers*, along with links to detailed web pages for the various drivers.

Note that not all drivers can be installed on all systems; Table 8.1 in *The Visualization Drivers* lists all the available drivers and the platforms on which they can be installed. For any of the visualization drivers to work, the corresponding graphics system must be installed beforehand.

Visualization drivers that do not depend on external libraries are by default incorporated into GEANT4 libraries during their installation. Here “installation of GEANT4 libraries” means the generation of GEANT4 libraries by compilation. The automatically incorporated visualization drivers are: DAWNFILE, HepRepFile, HepRepXML, RayTracer, VRML2FILE and ATree and GAGTree.

The OpenGL, Qt, OpenInventor, Qt3D, TSG, Vtk and RayTracerX drivers are not incorporated by default. These drivers must be selected when you build the GEANT4 Toolkit itself. This procedure is described in detail in the Installation Guide, to which you should refer.

### 8.2.2 How to Realize Visualization Drivers in an Executable

You can realize and use any of the visualization driver(s) you want in your GEANT4 executable, provided they are among the set installed beforehand into the GEANT4 libraries. A warning will appear if this is not the case.

In order to realize visualization drivers, you must instantiate and initialize a subclass of `G4VisManager` that implements the pure virtual function `RegisterGraphicsSystems()`. This subclass must be compiled in the user’s domain to force the loading of appropriate libraries in the right order. *The easiest way to do this is to use* `G4VisExecutive`:

```
auto visManager = new G4VisExecutive(argc, argv);
```

### 8.2.3 If you do wish to write your own subclass…

…you may do so. You will see how to do this by looking at `G4VisExecutive.icc`. A typical extract is:

```
...
  RegisterGraphicsSystem (new G4DAWNFILE);
...
#ifdef G4VIS_USE_OPENGLX
  RegisterGraphicsSystem (new G4OpenGLImmediateX);
  RegisterGraphicsSystem (new G4OpenGLStoredX);
#endif
...
```

If you wish to use `G4VisExecutive` but register an additional graphics system, `XXX` say, you may do so either before or after initializing:

```
visManager->RegisterGraphicsSystem(new XXX);
visManager->Initialize();
```

and add the library name to your `CMakeLists.txt` file:

```
target_link_libraries(exampleB1 G4visXXX ${Geant4_LIBRARIES})
```

## 8.2.4 By default...

...you get the DAWNFILE, HepRepFile, RayTracer, VRML2FILE and ATree drivers.

## 8.2.5 Optionally...

...you may install the OpenGL-Xlib, OpenGL-Motif, Qt, OpenInventor, Qt3D, TSG, Vtk and RayTracerX drivers, each of which can be enabled when configuring your application through CMake options - see Installation Guide.

For more details, see *The Visualization Drivers* and pages linked from there.

## 8.2.6 Visualization Manager

Visualization procedures are controlled by the "Visualization Manager", a class which must inherit from `G4VisManager` defined in the visualization category. Most users will find that they can just use the default visualization manager, `G4VisExecutive`. The Visualization Manager accepts users' requests for visualization, processes them, and passes the processed requirements to the abstract interface, i.e., to the currently selected visualization driver.

## 8.2.7 How to Write the `main()` Function

In order for your GEANT4 executable to perform visualization, you must instantiate and initialize "your" Visualization Manager in the `main()` function. The core of the Visualization Manager is the class `G4VisManager`, defined in the visualization category. This class requires that one pure virtual function be implemented, namely, `void RegisterGraphicsSystems()`. The easiest way to do this is to use `G4VisExecutive`, as described in Listing 8.1.

Listing 8.1: The form of the `main()` function.

```
//----- C++ source codes: Instantiation and initialization of G4VisManager
.....
// Your Visualization Manager
#include "G4VisExecutive.hh"
.....
// Instantiation and initialization of the Visualization Manager
G4VisManager* visManager = new G4VisExecutive;
// G4VisExecutive can take a verbosity argument - see /vis/verbose guidance.
// G4VisManager* visManager = new G4VisExecutive("Quiet");
visManager->Initialize();
.....
delete visManager;
//----- end of C++
```

*Do not forget* to delete the instantiated Visualization Manager.

Listing 8.2 shows a complete `main()` function (but see the examples distributed with GEANT4 for more ideas).

Listing 8.2: A `main()` function with interaction and visualization.

```
.....
#include "G4VisExecutive.hh"
#include "G4UIExecutive.hh"
.....
int main(int argc,char** argv)
{
  auto ui = new G4UIExecutive(argc, argv);

  auto* runManager = G4RunManagerFactory::CreateRunManager();

  runManager->SetUserInitialization(new DetectorConstruction);
  auto physicsList = new QBBC;
  runManager->SetUserInitialization(physicsList);
  runManager->SetUserInitialization(new ActionInitialization);

  auto visManager = new G4VisExecutive;
  visManager->Initialize();

  ui->SessionStart();

  delete visManager;
  delete runManager;
  delete ui;
}
```

The visualization manager prints useful information depending on the verbosity level:

```
Simple graded message scheme - give first letter or a digit:
 0) quiet,          // Nothing is printed.
 1) startup,        // Startup and endup messages are printed...
 2) errors,         // ...and errors...
 3) warnings,       // ...and warnings...
 4) confirmations, // ...and confirming messages...
 5) parameters,     // ...and parameters of scenes and views...
 6) all             // ...and everything available.
```

For example, in your `main()` function, write:

```
G4VisManager* visManager = new G4VisExecutive("Quiet");
```

or change with the `/vis/verbose` command.

## 8.3 The Visualization Drivers

As explained in the Introduction to Visualization, GEANT4 provides many different choices of visualization systems. Features and notes on each driver are briefly described here along with links to detailed web pages for the various drivers.

Details are given below for:

- *OpenGL*
- *Qt*
- *Open Inventor*
- *Open Inventor Extended Viewer*
- *Open Inventor Qt Viewer*
- *Qt3D*
- *ToolsSG*
- *VTK (Visualisation toolkit)*

- *HepRepFile*
- *DAWN*
- *VRML*
- *RayTracer*
- *gMocren*
- *ASCIITree*

## 8.3.1 Availability of drivers on the supported systems

Table 8.1 lists required graphics systems and supported platforms for the various visualization drivers. Please refer to the Installation Guide for details of how to build Geant4 with support for these drivers, and *Use of Geant4Config.cmake with find_package in CMake* for details on how to configure applications to use them.

Table 8.1: Required graphics systems and supported platforms for the various visualization drivers.

| Driver | Required Graphics System | Platform |
|---|---|---|
| OpenGL-Xlib | OpenGL | Linux, UNIX, Mac with Xlib |
| OpenGL-Motif | OpenGL | Linux, UNIX, Mac with Motif |
| OpenGL-Win32 | OpenGL | Windows |
| Qt | Qt, OpenGL | Linux, UNIX, Mac, Windows |
| OpenInventor-Qt | Open Inventor (Coin3D), Qt, OpenGL | Linux, UNIX, Mac |
| OpenInventor-X | Open Inventor (Coin3D), OpenGL | Linux, UNIX, Mac with Xlib and Motif |
| OpenInventor-X-Extended | Open Inventor (Coin3D), OpenGL | Linux, UNIX, Mac with Xlib and Motif |
| OpenInventor-Win32 | Open Inventor, OpenGL | Windows |
| Qt3D | Qt | Linux, UNIX, Mac, Windows |
| ToolsSG | OpenGL-ES, Qt or none (off-screen) | Linux, UNIX, Mac, Windows |
| VTK | VTK (vtk.org) | Linux, UNIX, Mac |
| VRML2FILE | Most internet browsers | Linux, UNIX, Mac, Windows |
| HepRepFile | HepRApp or FRED | Linux, UNIX, Mac, Windows |
| DAWNFILE | Fukui Renderer DAWN | Linux, UNIX, Mac, Windows |
| VRML2FILE | any VRML viewer | Linux, UNIX, Mac, Windows |
| RayTracer | any JPEG viewer | Linux, UNIX, Mac, Windows |
| RayTracerX | X11 (also produces a jpeg file) | Linux, UNIX, Mac. |
| ASCIITree | None | Linux, UNIX, Mac, Windows |

## 8.3.2 OpenGL

These drivers have been developed by John Allison and Andrew Walkden (University of Manchester). It is an interface to the de facto standard 3D graphics library, OpenGL. It is well suited for real-time fast visualization and demonstration. Fast visualisation is realized with hardware acceleration, reuse of shapes stored in a display list, etc.

Several versions of the OpenGL drivers are prepared. Versions for Xlib, Motif, Qt and Win32 platforms are available by default. For each version, there are two modes: immediate mode and stored mode. The former has no limitation on data size, and the latter is fast for visualizing large data repetitively, and so is suitable for animation.

Images can be exported using `/vis/ogl/export`.

More information can be found here: *How to save a view to an image file*

If you want to open a OGL viewer, the generic way is:

```
/vis/open OGL
```

According to your G4VIS_USE… variables it will open the correct viewer. By default, it will be open in stored mode. You can specify to open an "OGLS" or "OGLI" viewer, or even "OGLSXm","OGLIXm",… If you don't have Motif or Qt, all control is done from GEANT4 commands:

```
/vis/open OGLIX
/vis/viewer/set/viewpointThetaPhi 70 20
/vis/viewer/zoom 2
etc.
```

But if you have Motif libraries or Qt install, you can control GEANT4 from Motif widgets or mouse with Qt:

```
/vis/open OGLSQt
```

The OpenGL driver added Smooth shading and Transparency since GEANT4 release 8.0.

**Further information (OpenGL and Mesa):**

- https://www.opengl.org/
- https://www.mesa3d.org

### 8.3.3 Qt

This driver has been developed by Laurent Garnier (IN2P3, LAL Orsay). It is an interface to the powerful application framework, Qt, now free on most platforms. This driver also requires the OpenGL library.

The Qt driver is well suited for real-time fast visualization and demonstration. Fast visualization is realized with hardware acceleration, reuse of shapes stored in a display list, etc. All OpenGL features are implemented in the Qt driver, but one also gets mouse control of rotation/translation/zoom, the ability to save your scene in many formats (both vector and pixel graphics) and an easy interface for making movies.

Two display modes are available: Immediate mode and Stored mode. The former has no limitation on data size, and the latter is fast for visualizing large data repetitively, and so is suitable for animation.

This driver has the feature to open a vis window into the UI window as a new tab. You can have as many tabs you want and mix them from Stored or Immediate mode. To see the visualization window in the UI:

```
/vis/open OGL  (Generic way. For Stored mode if you have define your G4VIS_USE_QT variable)
or
/vis/open OGLI  (for Immediate mode)
or
/vis/open OGLS  (for Stored mode)
or
/vis/open OGLIQt  (for Immediate mode)
or
/vis/open OGLSQt  (for Stored mode)
```

**Further information (Qt):**

- Qt
- |Geant4| Visualization Tutorial using the Qt Driver

### 8.3.4 Open Inventor

The original drivers were developed by Jeff Kallenbach (FNAL) and Guy Barrand (IN2P3) based on the Hepvis class library originated by Joe Boudreau (Pittsburgh University). The Open Inventor drivers and the Hepvis class library are based on the well-established Open Inventor technology for scientific visualization. They have high extendibility. They support high interactivity, e.g., attribute editing of picked objects. Some Open Inventor viewers support "stereoscopic" effects.

It is also possible to save a visualized 3D scene as an OpenInventor-formatted file, and re-visualize the scene afterwards.

Because it is connected directly to the GEANT4 kernel, using same language as that kernel (C++), OpenInventor systems can have direct access to GEANT4 data (geometry, trajectories, etc.).

Because Open Inventor uses OpenGL for rendering, it supports lighting and transparency.

Open Inventor provides thumbwheel control to rotate and zoom.

Open Inventor supports picking to ask about data. [Control Clicking] on a volume turns on rendering of that volume's daughters. [Shift Clicking] a daughter turns that rendering off: If modeling opaque solid, effect is like opening a box to look inside.

**Further information (HEPVis and OpenScientist):**

- GEANT4 Inventor Visualization with OpenScientist
- Overall OpenScientist Home

**Further information (OpenInventor):**

- Josie Wernecke, "The Inventor Mentor", Addison Wesley (ISBN 0-201-62495-8)
- Josie Wernecke, "The Inventor Toolmaker", Addison Wesley (ISBN 0-201-62493-1)
- "The Open Inventor C++ Reference Manual", Addison Wesley (ISBN 0-201-62491-5)

### 8.3.5 Open Inventor Extended Viewer

This driver was developed by Rastislav Ondrasek, Pierre-Luc Gagnon and Frederick Jones (TRIUMF). It extends the functionality of the OpenInventor driver, described in the previous section, by adding a number of new features to the viewer.

Although this viewer is still available it has been superseded by the Open Inventor Qt Viewer (see below).

### 8.3.6 Open Inventor Qt Viewer

This driver was developed by Frederick Jones (TRIUMF) and is based in part on the Extended Viewer driver. It is supported on Linux/Unix/MacOS platforms and requires Qt5 and Coin3D libraries (Coin and SoQt) to be installed. When resources become available a Windows version of the driver will be pursued.

This is the preferred Open Inventor viewer and will potentially replace the older viewers described above. It incorporates all of their capabilities together with many added functions implemented via menu bar items, viewer buttons, a navigation panel, and keyboard and mouse inputs.

**Reference path navigation**

Most of the added features are concerned with navigation along a "reference path" which is a piecewise linear path through the geometry. The reference path can be any particle trajectory, which may be chosen at run time by selecting a trajectory with the mouse. Via Load and Save menu items in the File menu, a reference path can also be read from a file and the current reference path can be written to a file.

Once a reference path is established, the bottom part of the navigation panel is populated with a list of all elements in the geometry, ordered by their "distance" along the reference path (based on the perpendicular from the element center to the path).

**Navigation controls**

[L,R,U,D refer to the arrow keys on the keyboard]

- Select an element from the list: navigate along the path to the element's "location" (distance along the reference path).
- Shift-L and Shift-R: navigate to the previous or next element on the path (with wraparound).
- L and R: rotate 90 degrees around the perpendicular to the reference path
- U and D: rotate 90 degrees around the reference path
- Ctrl-L and Ctrl-R: rotate 90 degrees around the horizontal axis

All these keys have a "repeat" function for continuous motion.

The rotation keys put the camera in a definite orientation, whereas The Shift-L and Shift-R keys can be used to "fly" along the path in whatever camera orientation is in effect. NOTE: if this appears to be "stuck", try switching from orthonormal camera to perspective camera ("cube" viewer button).

Menu Items:

- Tools / Go to start of reference path: useful if you get lost
- Tools / Invert reference path: flips the direction of travel and the distance readout

**Reference path animation**

This is a special mode which flies the camera steadily along the path, without wraparound. The controls are:

- Tools Menu - Fly along Ref Path: start animation mode
- Page-Up: increase speed
- Page-Down: decrease speed
- U (arrow key): raise camera
- D (arrow key): lower camera
- ESC: exit animation mode

For suitable geometries the U and D keys can be used to get "Star Wars" style fly-over and fly-under effects.

**Bookmarks**

At any time, the viewpoint and other camera parameters can be saved in a file as a labelled "bookmark". The view can then be restored later in the current run or in another run. Bookmarks are displayed in a list in the top part of the navigation panel.

The default name for the bookmark file is "bookmarkFile" The first time a viewpoint is saved, this file will be created if it does not already exist. When the viewer is first opened, it will automatically read this file if present and load the viewpoints into the left-hand panel of the viewer's auxiliary window.

Controls:

- Select bookmark from list: restore this view
- Right-arrow VIEWER button: go to next bookmark
- Left-arrow VIEWER button: go to next bookmark
- "Floppy Disk" button: bookmark the current view. The user can type in a label for the bookmark, or use the default label provided.
- File Menu - Open Bookmark File: loads an existing bookmark file
- File Menu - New Bookmark File: creates a new bookmark file for saving subsequent views

**Special picking modes**

Controls:

- "Arrow +" VIEWER button: enable brief trajectory picking and mouse-over element readout For trajectories, the list of all trajectory points is replaced by the first and last point only, allowing easier identification of the particle without scrolling back. Passing the mouse over an element will give a readout of the volume name, material, and position on the reference path.
- "Crosshair" VIEWER button: select new reference path The cursor will change to a small cross (+) after which a trajectory can be selected to become the new reference path.

**Convenience feature**

When using the Open Inventor viewer with a terminal-based UI (e.g. tcsh) it is now possible to escape from the viewer without using the mouse.

In addition to the File - Escape menu item, pressing the "e" key on the keyboard will exit from the viewer's secondary event loop. The viewer will become inactive and control will return to the GEANT4 UI prompt.

### 8.3.7 Qt3D

As of writing for Release 11.1, Qt3D is an "experimental" driver exploiting a recently announced feature of Qt. Qt3D looks like Qt's attempt to get into visualisation as well as user interface. All the same, Qt programming is tough, so please bear with us. Please try it and give us feedback.

It has been developed so far by John Allison. The advantage, as we see it, is that it programs directly over Qt, which is then free to exploit the local system to its advantage. For example, on MacOS, it will in future (they say) build directly on Metal, making us independent of OpenGL (which Apple are deprecating). It is a way of future-proofing Geant4.

If you build with Qt, this driver will be instantiated automatically.

### 8.3.8 ToolsSG

Developed by Guy Barrand, this driver is based on his ToolsSG package distributed with Geant4 (which also supports the Geant4 analysis system). It offers the same features as the OpenGL drivers and comes with options over X11, Windows or Qt (depending on your CMake selections during Geant4 build - see Installation Guide). It has the ability to exploit local graphics systems so that, like Qt3D, it offers future-proofing for Geant4 visualisation.

Since Geant4 11.1, beside the X11, Windows, Qt "screen" sub-drivers, there is also the "offscreen" sub-driver permitting to produce file output at the png, jpeg, gl2ps formats by using only standalone C++ code based on the standard libraries (no need of extra graphical external packages, all the code comes within g4tools). This sub-driver is built by default and can be operated in a pure batch/offscreen program. In particular it permits to produce pictures at high resolution adequate for outreach.

ToolsSG also supports plotting. If you have registered histograms with the Geant4 analysis manager, they will be available for plotting at end of run. Example B5 illustrates how to do this.

These drivers use the scene graph logic found in the classes under:

```
source/externals/g4tools/include/tools/sg
```

(sg being for "scene graph").

These classes are themselves a subpart of the softinex/inlib and exlib thesaurus of code accumulated for long at Orsay (at LAL before 2020 and now at the IJCLab) to help doing visualization and data analysis for various projects. (The namespaces inlib and exlib had been changed to "tools" when importing classes within Geant4 to avoid clashes with apps using both Geant4 and straight softinex). This scene graph way of doing visualization is borrowed from the great OpenInventor developed by Silicon Graphics Incs in the 1980's. The idea is that a data representation is done by creating a scene graph which is a tree of "nodes". For example a tree has in general a first tools::sg::ortho (or sg::perspective) camera node specifying a camera projection (position, orientation and depth of view), some sg::matrix node permitting to position an object in a 3D space and then some shape nodes as sg::cube, sg::cylinder or sg::vertices (a node handling a set of points, lines, segments or triangles) used to represent a piece of detector or tracks.

Whence having built a scene graph, the rendering is done, typically after having received some expose event in a drawing area window, by applying a "render_action" that traverses the scene graph and asks to the nodes the actions that will be passed to a specific graphics engine. For example a shape node (cube, sphere, polyhedron), when traversed, will give to the render_action the graphics primitives (points, lines, segments, triangles) representing that shape. A camera node will give a projection matrix, a matrix node will give a model matrix. A common graphics engine being GL-ES, we have the tools::sg::GL_action class that does that for GL-ES on macOS, Linux and Windows. We have also various render_action to do offscreen rendering (gl2ps_action using gl2ps and zb_action to render in an in memory z-buffer). In softinex we have also a exlib::wasm::render to render in WebAssembly using WebGL and a exlib::metal::render to render within macOS/Cocoa/Metal, but these are not yet used in Geant4/vis.

In Geant4/vis, the ToolsSG directory contains code to create a viewer within various windowing systems, codes which are declared as "drivers" in the vis system. Today there are the TOOLSSG_QT_GLES and TOOLSSG_XT_GLES to create a viewing area ready for GL-ES rendering if using the GUI toolkits Qt or Xt/Motif (activated through the G4UIQt, G4UIXt classes), and TOOLSSG_X11_GLES, TOOLSSG_WINDOWS_GLES to create a GL-ES viewing area straight on X11 or Windows windowing systems.

From a user point of view, typical commands to create a ToolsSG viewer are:

```
/vis/sceneHandler/create TSG scene-handler-tsg
/vis/viewer/create scene-handler-tsg viewer-tsg 600x600-0+0
```

or with the compound command:

```
/vis/viewer/open TSG 600x600-0+0
```

Someone can specify straight a TOOLSSG_[QT,XT,X11,WINDOWS]_GLES name driver, but if specifying "TSG", the G4/vis system will pick the "right one", according to the kind of GUI or windowing context which is choosen (in general Qt for now).

Obviously, these drivers must have been built when building/installing Geant4. With the G4 cmake system, this is done by specifying the cmake flag:

```
-DGEANT4_USE_TOOLSSG=XX
```

where XX is one of: OFF, X11, XT, QT or WIN32.

In the case of "offscreen", someone has to open with:

```
/vis/viewer/open TSG_OFFSCREEN 600x600
```

The given size will be the size widthxheight in pixels of the picture in the output file.

When a ToolsSG viewer is created, vis commands triggering the representation of a piece of detector or a track are the same as for other drivers. For example, as can be found in the examples/basic/B1/vis.mac:

```
# Draw geometry:
/vis/drawVolume
...
# Draw smooth trajectories at end of event:
/vis/scene/add/trajectories smooth
...
```

On a technical point of view, the G4ToolsSceneHandler class is the place where tools::sg nodes are created according each Geant4/vis primitive type (G4Polyhedron, G4Polyline, G4Text, etc…).

**ToolsSG /vis/tsg specific commands:**

The:

```
/vis/tsg/export
```

permits to write the content of the current ToolsSG "screen" viewer in a file at various formats. Default file is out.eps and default format is gl2ps_eps. Today, available formats are:

```
gl2ps_eps: gl2ps producing eps
gl2ps_ps:  gl2ps producing ps
gl2ps_pdf: gl2ps producing pdf
gl2ps_svg: gl2ps producing svg
gl2ps_tex: gl2ps producing tex
gl2ps_pgf: gl2ps producing pgf
zb_ps: tools::sg offscreen zbuffer put in a PostScript file.
```

An example of usage is:

```
/vis/tsg/export gl2ps_pdf out.pdf
```

Another command is:

```
/vis/tsg/plotter/printParameters
```

It permits to print the available keys used to customize a ToolsSG plotter. This command is more documented in the ToolsSG plotting section.

**ToolsSG /vis/tsg/offscreen specific commands:**

These commands are available when the current viewer is a TSG_OFFSCREEN one. In this case the default file format is zb_png. The picture is produced with:

```
/vis/viewer/rebuild
```

by using the tools::sg offscreen zbuffer, and is put in a png file with the tools::fpng png file writer.

The default file name is:

```
g4tsg_offscreen_[format]_[index].[suffix]
```

with:

```
index: starting at one and incremented at each file production.
format:
  zb_png: tools::sg offscreen zbuffer put in a png file.
  zb_jpeg: tools::sg offscreen zbuffer put in a jpeg file.
  zb_ps: tools::sg offscreen zbuffer put in a PostScript file.
  gl2ps_eps: gl2ps producing eps
  gl2ps_ps:  gl2ps producing ps
  gl2ps_pdf: gl2ps producing pdf
  gl2ps_svg: gl2ps producing svg
  gl2ps_tex: gl2ps producing tex
  gl2ps_pgf: gl2ps producing pgf
suffix: according to the choosen file format: eps, ps, pdf, svg, tex, pgf, png, jpeg.
```

You can change the file name with:

```
/vis/tsg/offscreen/set/file <file name>
```

You can change the automatic file name construction with:

```
/vis/tsg/offscreen/set/file auto <prefix> <true|false to reset the index>
```

The default picture size, in pixels, is the one given when doing a:

```
/vis/open TSG_OFFSCREEN [width]x[height]
```

for example:

```
/vis/open TSG_OFFSCREEN 1200x1200
```

or by taking the default Geant visualization system viewer size (600x600):

```
/vis/open TSG_OFFSCREEN
```

But you can change it after an “open” with:

```
/vis/tsg/offscreen/set/size <width> <height>
```

We remember that after having opened a ToolsSG offscreen viewer, you have to do an explicit:

```
/vis/viewer/rebuild
```

to produce a file.

About the picture size, note that the gl2ps files will grow with the number of primitives (gl2ps does not have a zbuffer logic). The “zb” files will not grow with the number of primitives, but with the size of the viewer. It should be preferred for scenes with a lot of objects to render. With zb, to have a better rendering, do not hesitate to have a large viewer size.

About transparency, the zb formats handle it. The gl2ps formats don’t, in this case you can use:

```
/vis/tsg/offscreen/set/transparency false
```

to not draw the transparent objects.

To have a starting point, go in examples/basic/B1 and play with the tsg_offscreen.mac macro file to see how to operate all these.

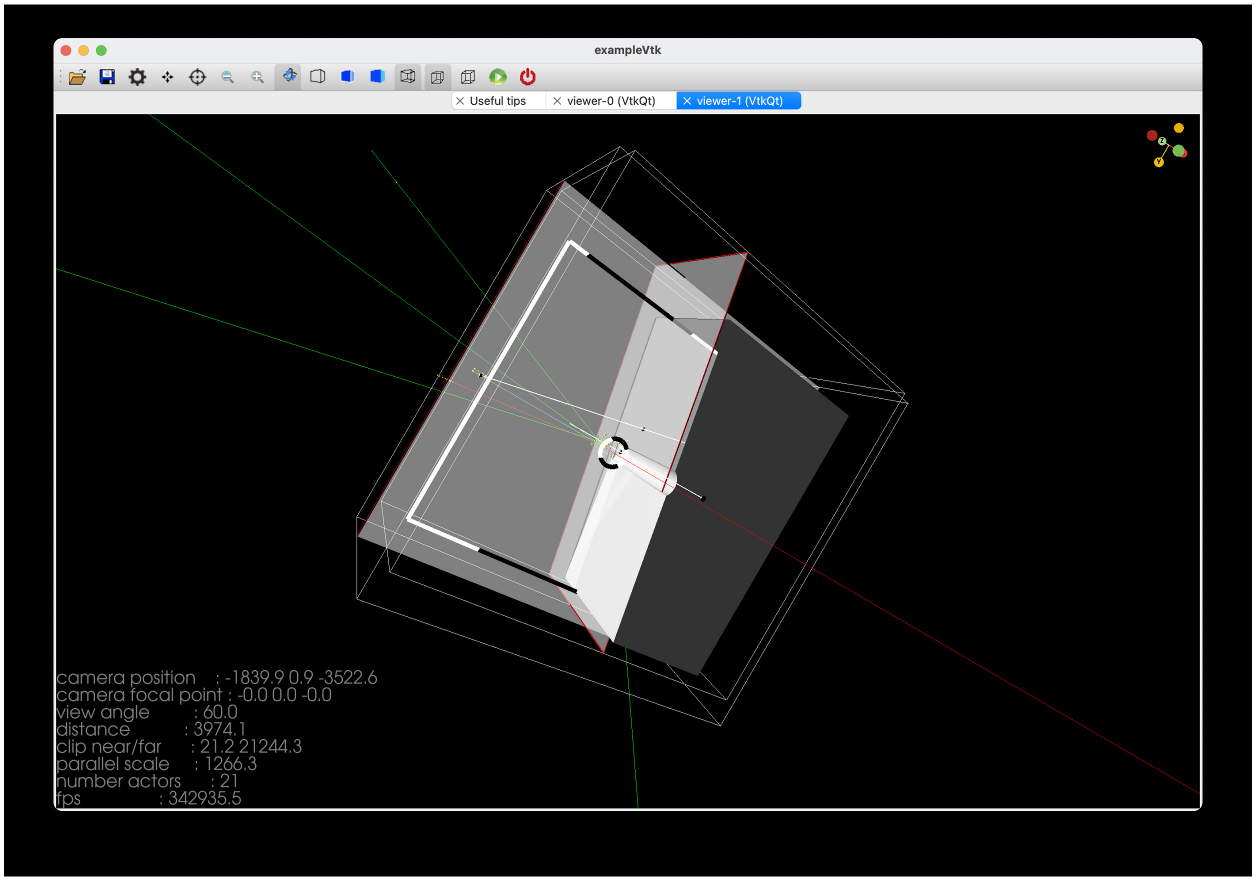
### 8.3.9 VTK (Visualisation toolkit)

Example of VtkQt visualisation driver

Developed by Stewart Boogert (University of Manchester) and Laurie Nevay (CERN), this driver exploits the Visualisation Toolkit VTK (http://vtk.org). Its focus is high performance, pipelined (deferred) and instanced rendering. You need to install the VTK libraries - see Installation Guide. There is a “native” driver, but also one which melds with Qt if you also have Qt installed.3 viewers are currently available based on VTK

| VTK viewer | Description |
|---|---|
| VtkNative | Native viewer (open Cococa, Xwindows, Microsoft windows window |
| VtQt | Integrated with Qt |
| VtkOffscreen | Graphics started but no window open |

VTK allows for some functionality not common to the other visualisation systems, the available commands are



```
camera position    : -1839.9 0.9 -3522.6
camera focal point : -0.0 0.0 -0.0
view angle         : 60.0
distance           : 3974.1
clip near/far      : 21.2 21244.3
parallel scale     : 1266.3
number actors      : 21
fps                : 342935.5
```

| VTK specific command | Description |
|---|---|
| /vis/vtk/add/imageOverlay | Place an 2D image in scene |
| /vis/vtk/add/geometryOverlay | Place an 3D data in scene |
| /vis/vtk/set/clipper | Add interactive clipper |
| /vis/vtk/set/cutter | Add interactive cutter |
| /vis/vtk/set/hud | Set Head up display (0,1) |
| /vis/vtk/set/polyhedronPipeline | Set polyhedron pipeline type (separate, append, bake, tensor) |
| /vis/vtk/set/shadows | Enable/disable shadows |
| /vis/vtk/set/warnings | Enable/disable VTK warnings |
| /vis/vtk/export | Export scene as VTP/VTU/VRML/GLTF/OBJ |
| /vis/vtk/exportCutter | Export cutters VTP/VTU |
| /vis/vtk/printDebug | Print information on pipelines |
| /vis/vtk/startInteraction | Start VtkNative window interaction |

A pipeline is a sequence of steps which converts a data structure into a set of graphics operations. The current driver implements a handful of pipelines to create a visualisation similar to that produced by OpenGL. Simple pipelines can be added to the VTK driver without understanding the entire visualisation system, for example a `G4Polyhedron` to `vtkActor` is:

```
polydataPoints   = vtkSmartPointer<vtkPoints>::New();
polydataCells    = vtkSmartPointer<vtkCellArray>::New();
polydata         = vtkSmartPointer<vtkPolyData>::New();

polydata->SetPoints(polydataPoints);
polydata->SetPolys(polydataCells);

// clean input polydata
auto filterClean = vtkSmartPointer<vtkCleanPolyData>::New();
filterClean->PointMergingOn();
filterClean->AddInputData(polydata);
AddFilter(filterClean);

// ensure triangular mesh
auto filterTriangle = vtkSmartPointer<vtkTriangleFilter>::New();
filterTriangle->SetInputConnection(filterClean->GetOutputPort());
AddFilter(filterTriangle);

// calculate normals with a feature angle of 45 degrees
auto filterNormals = vtkSmartPointer<vtkPolyDataNormals>::New();
filterNormals->SetFeatureAngle(45);
filterNormals->SetInputConnection(filterTriangle->GetOutputPort());
AddFilter(filterNormals);

// mapper
mapper = vtkSmartPointer<vtkPolyDataMapper>::New();
mapper->SetInputConnection(GetFinalFilter()->GetOutputPort());
mapper->SetColorModeToDirectScalars();

// add to actor
actor = vtkSmartPointer<vtkActor>::New();
actor->SetMapper(mapper);
actor->SetVisibility(1);
```

Very complex algorithms can be added into a pipeline without a user being an expert in 3D graphics programming. `G4VVtkPipeline` the base class for a pipeline can be chained together so once a visualisation pipeline is written it can be reused quickly. Pipelines can be added and removed on the fly without changing other pipelines so the graphics scene does not need to be rebuilt. The current core pipelines are

| Pipeline | Description | Geant4 primitive |
|---|---|---|
| `G4VVtkPipeline` | Base class for pipelines | |
| `G4VtkPolydataPipeline` | Class for 3D mesh data | |
| `G4VtkPolydataPolylinePipeline` | Class for 3D polyline | `G4Polyline` and `G4Square` |
| `G4VtkPolydataSpherePipeline` | Class for 3D sphere | `G4Circle` |
| `G4VtkPolydataPolyline2DPipeline` | Class for 2D polyline | `G4Polyline` |
| `G4VtkPolydataInstancePipeline` | Class for 3D mesh data with instances | `G4Polyhedron` |
| `G4VtkPolydataInstanceAppendPipeline` | Class for 3D mesh data with instances | `G4Polyhedron` |
| `G4VtkPolydataInstanceBakePipeline` | Class for 3D mesh data with instances | `G4Polyhedron` |
| `G4VtkPolydataInstanceTensorPipeline` | Class for 3D mesh data with instances | `G4Polyhedron` |
| `G4VtkText2DPipeline` | Class for 2D text | `G4Text` |
| `G4VtkTextPipeline` | Class for 3D text | `G4Text` |

Pipelines which can be chained onto a `G4VVtkPipeline`.

| Pipeline | Description |
|---|---|
| `G4VtkCutterPipeline` | Cut geometry through plane |
| `G4VtkClipOpenPipeline` | Clip (remove) geometry through plane |
| `G4VtkClipClosedSurfacePipeline` | Clip (remove) geometry through plane |

These pipelines are constructed in such a way that any pipeline can be cut or clipped.

### VTK window interaction

These are specific to the VTK window and do not change the Geant4 viewing system

| Key | Action |
|---|---|
| `s` | Surface |
| `w` | Wireframe |
| `3` | Toggle red/blue stereo |
| `q` | Exit interaction (VtkNative only) |

**Note:** To interact with the VtkNative viewer with shell UI the command `/vis/vtk/startInteraction` needs to be issued. This will block mouse and keyboard interaction with the G4 UI and input will be passed to Vtk. Once interaction is no longer required hit `q`.

**Note:** There are compatibility issues with graphics drivers that use different versions of OpenGL, so for the time being (Geant4 11.2), selecting Vtk suppresses the OpenGL drivers and ToolsSG drivers that use GLES.

### VTK Advanced features

VTK allows many possibilities beyond those already implemented in Geant4.

1. XR (VR/AR) integration
2. Shadows
3. Physically based rendering
4. Camera motion blur

## 8.3.10 HepRepFile

The HepRepFile driver creates a HepRep XML file in the HepRep1 format suitable for viewing with the HepRApp HepRep Browser.

The HepRep graphics format is further described at http://www.slac.stanford.edu/~perl/heprep .

To write just the detector geometry to this file, use the command:

```
/vis/viewer/flush
```

Or, to also include trajectories and hits (after the appropriate /vis/viewer/add/trajectories or /vis/viewer/add/hits commands), just issue:

```
/run/beamOn 1
```

HepRepFile will write a file called G4Data0.heprep to the current directory. Each subsequent file will have a file name like G4Data1.heprep, G4Data2.heprep, etc.

View the file using the HepRApp HepRep Browser, available from:

http://www.slac.stanford.edu/~perl/HepRApp/ .

HepRApp allows you to pick on volumes, trajectories and hits to find out their associated HepRep Attributes, such as volume name, particle ID, momentum, etc. These same attributes can be displayed as labels on the relevant objects, and you can make visibility cuts based on these attributes ("show me only the photons", or "omit any volumes made of iron").

HepRApp can read heprep files in zipped format as well as unzipped, so you can save space by applying gzip to the heprep file. This will reduce the file to about five percent of its original size.

Several commands are available to override some of HepRepFile's defaults

- You can specify a different directory for the heprep output files by using the setFileDir command, as in:

  ```
  /vis/heprep/setFileDir <someOtherDir/someOtherSubDir>
  ```

- You can specify a different file name (the part before the number) by using the setFileName command, as in:

  ```
  /vis/heprep/setFileName <my_file_name>
  ```

  which will produce files named <my_file_name>0.heprep, <my_file_name>1.heprep, etc.
- You can specify that each file should overwrite the previous file (always rewriting to the same file name) by using the setOverwrite command, as in:

  ```
  /vis/heprep/setOverwrite true
  ```

  This may be useful in some automated applications where you always want to see the latest output file in the same location.
- GEANT4 visualization supports a concept called "culling", by which certain parts of the detector can be made invisible. Since you may want to control visibility from the HepRep browser, turning on visibility of detector parts that had defaulted to be invisible, the HepRepFile driver does not omit these invisible detector parts from

the HepRep file. But for very large files, if you know that you will never want to make these parts visible, you can choose to have them left entirely out of the file. Use the /vis/heprep/setCullInvisibles command, as in:

```
/vis/heprep/setCullInvisibles true
```

**Further information:**

- HepRApp Users Home Page: http://www.slac.stanford.edu/~perl/HepRApp/
- HepRep graphics format: http://www.slac.stanford.edu/~perl/heprep

### 8.3.11 DAWN

The DAWN drivers are interfaces to Fukui Renderer DAWN, which has been developed by Satoshi Tanaka, Minato Kawaguti et al (Fukui University). It is a vectorized 3D PostScript processor, and so well suited to prepare technical high quality outputs for presentation and/or documentation. It is also useful for precise debugging of detector geometry. Remote visualization, off-line re-visualization, cut view, and many other useful functions of detector simulation are supported. A DAWN process is automatically invoked as a co-process of GEANT4 when visualization is performed, and 3D data are passed with inter-process communication, via a file.

When GEANT4 Visualization is performed with the DAWN driver, the visualized view is automatically saved to a file named `g4.eps` in the current directory, which describes a vectorized (Encapsulated) PostScript data of the view.

There are two kinds of DAWN drivers, the DAWNFILE driver and the DAWN-Network driver. The DAWNFILE driver is usually recommended, since it is faster and safer in the sense that it is not affected by network conditions.

The DAWNFILE driver sends 3D data to DAWN via an intermediate file, named `g4.prim` in the current directory. The file `g4.prim` can be re-visualized later without the help of GEANT4. This is done by invoking DAWN by hand:

```
% dawn g4.prim
```

DAWN files can also serve as input to two additional programs:

- A standalone program, DAWNCUT, can perform a planar cut on a DAWN image. DAWNCUT takes as input a .prim file and some cut parameters. Its output is a new .prim file to which the cut has been applied.
- Another standalone program, DAVID, can show you any volume overlap errors in your geometry. DAVID takes as input a .prim file and outputs a new .prim file in which overlapping volumes have been highlighted. The use of DAVID is described in section *Detecting Overlapping Volumes* of this manual.

### 8.3.12 VRML

These drivers were developed by Satoshi Tanaka and Yasuhide Sawada (Fukui University). They generate VRML files, which describe 3D scenes to be visualized with a proper VRML viewer, at either a local or a remote host. It realizes virtual-reality visualization with your WWW browser. There are many excellent VRML viewers, which enable one to perform interactive spinning of detectors, walking and/or flying inside detectors or particle showers, interactive investigation of detailed detector geometry etc.

The VRML2FILE driver sends 3D data to your VRML viewer, which is running on the same host machine as GEANT4, via an intermediate file named `g4.wrl` created in the current directory. This file can be re-visualization afterwards. In visualization, the name of the VRML viewer should be specified by setting the environment variable `G4VRML_VIEWER` beforehand. For example,

```
% setenv G4VRML_VIEWER  "netscape"
```

Its default value is `NONE`, which means that no viewer is invoked and only the file `g4.wrl` is generated.

### 8.3.13 RayTracer

This driver was developed by Makoto Asai and Minamimoto (Hirosihma Instutute of Technology). It performs ray-tracing visualization using the tracking routines of GEANT4. It is, therefore, available for every kinds of shapes/solids which GEANT4 can handle. It is also utilized for debugging the user's geometry for the tracking routines of GEANT4. It is well suited for photo-realistic high quality output for presentation, and for intuitive debugging of detector geometry. It produces a JPEG file. This driver is by default listed in the available visualization drivers of user's application.

Some pieces of geometries may fail to show up in other visualization drivers (due to algorithms those drivers use to compute visualizable shapes and polygons), but RayTracer can handle any geometry that the GEANT4 navigator can handle.

Because RayTracer in essence takes over GEANT4's tracking routines for its own use, RayTracer cannot be used to visualize Trajectories or hits.

An X-Window version, called RayTracerX, can be selected by setting `GEANT4_USE_RAYTRACER_X11` (for CMake) at GEANT4 library build time and application (user code) build time (assuming you use the standard visualization manager, `G4VisExecutive`, or an equally smart vis manager). RayTracerX builds the same jpeg file as RayTracer, but simultaneously renders to screen so you can watch as rendering grows progressively smoother.

RayTracer has its own built-in commands - `/vis/rayTracer/...`. Alternatively, you can treat it as a normal vis system and use `/vis/viewer/...` commands, e.g:

```
/vis/open RayTracerX
/vis/drawVolume
/vis/viewer/set/viewpointThetaPhi 30 30
/vis/viewer/refresh
```

The view parameters are translated into the necessary RayTracer parameters.

RayTracer is compute intensive. If you are unsure of a good viewing angle or zoom factor, you might be advised to choose them with a faster renderer, such as OpenGL. Then, on opening the RayTracer, it will pick up the current view parameters.:

```
/vis/open OGL
/vis/drawVolume
/vis/viewer/zoom  # plus any /vis/viewer/commands that get you the view you want.
/vis/open RayTracerX  # or RayTracer
/vis/viewer/refresh
```

### 8.3.14 gMocren

The gMocrenFile driver creates a gdd file suitable for viewing with the gMocren volume visualizer. gMocren, a sophisticated tool for rendering volume data, can show volume data such as GEANT4 dose distributions overlaid with scoring grids, trajectories and detector geometry. gMocren provides additional advanced functionality such as transfer functions, colormap editing, image rotation, image scaling, and image clipping.

gMocren is further described at http://geant4.kek.jp/gMocren/. At this link you will find the gMocren download, the user manual, a tutorial and some example gdd data files.

Please note that the gMocren file driver is currently considered a Beta release. Users are encouraged to try this driver, and feedback is welcome, but users should be aware that features of this driver may change in upcoming releases.

To send volume data from GEANT4 scoring to a gMocren file, the user needs to tell the gMocren driver the name of the specific scoring volume that is to be displayed. For scoring done in C++, this is the name of the sensitive volume. For command-based scoring, this is the name of the scoring mesh.

```
/vis/gMocren/setVolumeName <volume_name>
```

The following is an example of the minimum command sequence to send command-based scoring data to the a gMocren file:

```
# an example of a command-based scoring definition
/score/create/boxMesh scoringMesh       # name of the scoring mesh
/score/mesh/boxSize 10. 10. 10. cm      # dimension of the scoring mesh
/score/mesh/nBin 10 10 10               # number of divisions of the scoring mesh
/score/quantity/energyDeposit eDep      # quantity to be scored
/score/close
# configuration of the gMocren-file driver
/vis/scene/create
/vis/open gMocrenFile
/vis/gMocren/setVolumeName scoringMesh
```

To add detector geometry to this file:

```
/vis/viewer/flush
```

To add trajectories and primitive scorer hits to this file:

```
/vis/scene/add/trajectories
/vis/scene/add/pshits
/run/beamOn 1
```

gMocrenFile will write a file named G4_00.gd to the current directory. Subsequent draws will create files named g4_01.gdd, g4_02.gdd, etc. An alternate output directory can be specified with an environment variable:

```
export G4GMocrenFile_DEST_DIR=<someOtherDir/someOtherSubDir/>
```

View the resulting gMocren files with the gMocren viewer, available from: http://geant4.kek.jp/gMocren/.

## 8.3.15 Visualization of detector geometry tree

ASCIITREE is a visualization driver that is not actually graphical but that dumps the volume hierarchy as a simple text tree using `/vis/drawTree`.

ASCIITree has command to control its verbosity, `/vis/ASCIITree/verbose`. The verbosity value controls the amount of information available, e.g., physical volume name alone, or also logical volume and solid names. If the volume is "sensitive" and/or has a "readout geometry", this may also be indicated. Also, the mass of the physical volume tree(s) can be printed (but beware - higher verbosity levels can be computationally intensive).

At verbosity level 4, ASCIITree calculates the mass of the complete geometry tree taking into account daughters up to the depth specified for each physical volume. The calculation involves subtracting the mass of that part of the mother that is occupied by each daughter and then adding the mass of the daughter, and so on down the hierarchy.

```
/vis/ASCIITree/Verbose 4
/vis/viewer/flush
"HadCalorimeterPhysical":0 / "HadCalorimeterLogical" / "HadCalorimeterBox"(G4Box),
                             1.8 m3 , 11.35 g/cm3
"HadCalColumnPhysical":-1 (10 replicas) / "HadCalColumnLogical" / "HadCalColumnBox"(G4Box),
                                          180000 cm3, 11.35 g/cm3
"HadCalCellPhysical":-1 (2 replicas) / "HadCalCellLogical" / "HadCalCellBox"(G4Box),
                                       90000 cm3, 11.35 g/cm3
"HadCalLayerPhysical":-1 (20 replicas) / "HadCalLayerLogical" / "HadCalLayerBox"(G4Box),
                                         4500 cm3, 11.35 g/cm3
"HadCalScintiPhysical":0 / "HadCalScintiLogical" / "HadCalScintiBox"(G4Box),
                           900 cm3, 1.032 g/cm3

Calculating mass(es)...
Overall volume of "worldPhysical":0, is 2400 m3
Mass of tree to unlimited depth is 22260.5 kg
```

Some more examples of ASCIITree in action:

```
Idle> /vis/ASCIITree/verbose 1
Idle> /vis/drawTree
#  Set verbosity with "/vis/ASCIITree/verbose "
#    <  10: - does not print daughters of repeated placements, does not repeat replicas.
#    >= 10: prints all physical volumes.
#  The level of detail is given by verbosity%10:
#  for each volume:
#    >=  0: physical volume name.
#    >=  1: logical volume name (and names of sensitive detector and readout geometry, if any).
#    >=  2: solid name and type.
#    >=  3: volume and density.
#    >=  5: daughter-subtracted volume and mass.
#  and in the summary at the end of printing:
#    >=  4: daughter-included mass of top physical volume(s) in scene to depth specified.
.....
"Calorimeter", copy no. 0, belongs to logical volume "Calorimeter"
  "Layer", copy no. -1, belongs to logical volume "Layer" (10 replicas)
    "Absorber", copy no. 0, belongs to logical volume "Absorber"
      "Gap", copy no. 0, belongs to logical volume "Gap"
.....
Idle> /vis/ASCIITree/verbose 15
Idle> /vis/drawTree
....
 "tube_phys":0 / "tube_L" / "tube"(G4Tubs), 395841 cm3, 1.782 mg/cm3,
                          9.6539e-08 mm3, 1.72032e-10 mg
   "divided_tube_phys":0 / "divided_tube_L" / "divided_tube"(G4Tubs), 65973.4 cm3,
                          1.782 mg/cm3, 7587.54 cm3, 13.521 g
     "divided_tube_inset_phys":0 / "divided_tube_inset_L" / "divided_tube_inset"(G4Tubs),
                                  58385.9 cm3, 1.782 mg/cm3, 6.03369e-09 mm3, 1.0752e-11 mg
       "sub_divided_tube_phys":0 / "sub_divided_tube_L" / "sub_divided_tube"(G4Tubs),
                                  14596.5 cm3, 1.782 mg/cm3, 12196.5 cm3, 21.7341 g
.....
Calculating mass(es)...
Overall volume of "expHall_P":0, is 8000 m3  and the daughter-included mass to unlimited depth
                                 is 78414 kg
.....
```

For the complete list of commands and options, see the Control...UICommands section of this user guide.

## 8.4 Controlling Visualization from Commands

This section describes just a few of the more commonly used visualization commands. For the complete list of commands and options, see the Control...UICommands section of this user guide.

These commands can by typed on the session command line when in Idle state:

```
Idle> /vis/drawVolume
```

or specified in a macro file that is executed from the command line:

```
Idle> /control/execute vis.mac
```

or from your application, e.g.:

```
UImanager->ApplyCommand("/control/execute vis.mac");
```

Most examples have a `vis.mac` file, where you may look for inspiration.

**Warning:** This section is not a complete description of all visualisation commands; they are too numerous and continually evolving. Please refer to the command guidance, Control…UICommands or simply type "ls vis" or "help". Some viewers, notably Qt, offer interactivie guidance under the "Help" menu.

## 8.4.1 Scene, scene handler, and viewer

In using the visualization commands, it is useful to know the concept of "scene", "scene handler", and "viewer". A "scene" is a set of visualizable raw 3D data. A "scene handler" is a graphics-data modeler, which processes raw data in a scene for later visualization. And a "viewer" generates images based on data processed by a scene handler. Roughly speaking, a set of a scene handler and a viewer corresponds to a visualization driver.

The steps of performing GEANT4 visualization are explained below, though some of these steps may be done for you so that in practice you may use as few as just two commands (such as `/vis/open` plus `/vis/drawVolume`). The seven steps of visualization are:

Table 8.2: Seven steps of visualization.

| Step | | Command | Alternative command |
|---|---|---|---|
| 1 | Create a scene handler and a viewer | /vis/sceneHandler/create /vis/viewer/create | /vis/open |
| 2 | Create an empty scene | /vis/scene/create | /vis/drawVolume |
| 3 | Add raw 3D data to the created scene | /vis/scene/add/volume | |
| 4 | Attach the current scene to the current scene handler | /vis/sceneHandler/attach | |
| 5 | Set camera parameters, drawing style (wireframe/surface), etc | E.g., /vis/viewer/set/viewpoint | |
| 6 | Make the viewer execute visualization | /vis/viewer/refresh | |
| 7 | Declare the end of visualization for flushing | /vis/viewer/flush | |

For details about the commands, see below.

These seven steps can be controlled explicitly to create multiple scenes and multiple viewers, each with its own set of parameters, with easy switching from one scene to another. But for the most common case of just having one scene and one viewer, many steps are handled implicitly for you.

## 8.4.2 Choosing a graphics viewer: `/vis/open` command

Command "`/vis/open`" creates a scene handler and a viewer, which corresponds to Step 1.

**Command:** `/vis/open [<driver_tag_name>]`

- **Optional argument ``<driver_tag_name>``**
  We recommend you omit this and choose your driver at run time. In that case a driver may be chosen:
  - by argument in G4VisExecutive construction.
  - by environment variable, G4VIS_DEFAULT_DRIVER.
  - by information in ~/.g4session.
  - If you do not use any of the above, a driver will be chosen by mode (batch/interactive) and by your build flags.

  When using environment variable G4VIS_DEFAULT_DRIVER, the format is `<graphics-system> [<window-size-hint>]`, e.g:

```
export G4VIS_DEFAULT_DRIVER=OGL
setenv G4VIS_DEFAULT_DRIVER OI
export G4VIS_DEFAULT_DRIVER="TSG_OFFSCREEN 1200x1200"
```

then simply execute your application:

```
./<your-application>
```

or, to set the environment temporarily and exclusively for your application:

```
G4VIS_DEFAULT_DRIVER=Vtk ./<your-application>
```

Using `~/.g4session` (a file `.g4session` in your home directory), the first line is the default UI session. Subsequent lines have the format:

```
<your-app-name> <ui-session> [<vis-driver>] [<window-size-hint]
```

For example:

```
Qt  # Default session
#exampleB1 tcsh
exampleB1 Qt TSG 1000x1000+0-0
```

For a list of possible drivers see list of registered graphics systems printed at the start of execution

- **Action**
  Create a visualization driver, i.e. a scene handler and a viewer.

To see a list of driver_tag_names:

```
/vis/list
```

(produces a lot of information) or:

```
/vis/open xx
```

which produces:

```
parameter value (xx) is not listed in the candidate List.
Candidates are: ATree DAWNFILE HepRepFile HepRepXML OGL OGLI OGLIQt OGLS OGLSQt RayTracer␣
↪VRML1FILE VRML2FILE gMocrenFile
```

For additional options, see the Control. . . UICommands section of this user guide.

### 8.4.3 Create an empty scene: `/vis/scene/create` command

Command "`/vis/scene/create`" creates an empty scene, which corresponds to Step 2.

**Command:** `/vis/scene/create [scene_name]`

- **Argument**
  A name for this scene. Created for you if you don't specify one.

### 8.4.4 Visualization of a physical volume: `/vis/drawVolume` command

`/vis/drawVolume` is a "compound" command that creates a new scene (`/vis/scene/create`), adds a volume (`/vis/scene/add/volume`) and attaches it (`/vis/sceneHandler/attach`) to the current viewer (`/control/verbose 2` to see all the invoked commands). It takes care of steps 2, 3, 4 and 6. Command `/vis/viewer/flush` may be required in order to do the final Step 7.

**Commands:**

```
/vis/drawVolume [physical-volume-name]
```

- **Argument**
  If physical-volume-name is "world" (the default), the top of the main geometry tree (material world) is added. If "worlds", the tops of all worlds - material world and parallel worlds, if any - are added. Otherwise a search of all worlds is made.
  In the last case the names of all volumes in all worlds are matched against physical-volume-name. If this is of the form "/regexp/", where regexp is a regular expression (see C++ regex), the match uses the usual rules of regular expression matching. Otherwise an exact match is required.
  For example, "/Shap/" matches "Shape1" and "Shape2".
- **Action**
  Creates a scene consisting of the given physical volume(s) and asks the current viewer to draw it. The scene becomes current. Command "`/vis/viewer/flush`" should follow this command in order to declare end of visualization.
- **Example: Visualization of the whole world with coordinate axes**

  ```
  /vis/drawVolume
  /vis/scene/add/axes 0 0 0 500 mm
  /vis/viewer/flush
  ```

### 8.4.5 Visualization of a parameterised volume

The above command `/vis/drawVolume` works fine, but with parameterisation (see *Advanced parameterisations for 'nested' parameterised volumes*) you can get a very large number of volumes that can overwhelm a graphics system. The commands:

```
/vis/viewer/set/specialMeshRendering
```

and, optionally, the following:

```
/vis/viewer/set/specialMeshRenderingOption
/vis/viewer/set/specialMeshVolumes
```

can greatly improve performance and visual clarity.

### 8.4.6 Visualization of a logical volume: `/vis/drawLogicalVolume` command

`/vis/drawLogicalVolume` is a "compound" command that creates a new scene (`/vis/scene/create`), adds a logical volume (`/vis/scene/add/logicalVolume`) and attaches it (`/vis/sceneHandler/attach`) to the current viewer (`/control/verbose 2` to see all the invoked commands). It shows all that can be visualised about a logical volume—Booleans, voxels, readout geometries and overlaps—and adds axes in the local coordinate system. All options are on by default.

This command is synonymous with `/vis/specify`.

**Command:** `vis/drawLogicalVolume <logical-volume-name> [<depth-of-descent>] [<booleans-flag>] [<voxels-flag>] [<readout-flag>] [<axes-flag>] [<check-overlap-flag>]`

- **Argument**
  A logical-volume name.
- **Action**
  Creates a scene consisting of the given logical volume and asks the current viewer to draw it. The scene becomes current.
- **Example (visualization of a selected logical volume with coordinate axes)**

```
/vis/drawLogicalVolume Absorber
/vis/scene/add/text 0 0 0 mm 40 -100 -200 LogVol:Absorber
/vis/viewer/flush
```

For more information, use the `help` facility or refer to Control. . . UICommands.

### 8.4.7 Visualization of trajectories: `/vis/scene/add/trajectories` command

Command "`/vis/scene/add/trajectories [smooth] [rich]`" adds trajectories to the current scene. The optional parameters "smooth" and/or "rich" (you may specify either, neither or both) invoke, if "smooth" is specified, the storing and displaying of extra points on curved trajectories and, if "rich" is specified, the storing, for possible subsequent selection and display, of additional information, such as volume names, creator process, energy deposited, global time. Be aware, of course, that this imposes computational and memory overheads. Note that this automatically issues the appropriate "`/tracking/storeTrajectory`" command so that trajectories are stored (by default they are not). The visualization is performed with the command "`/run/beamOn`" unless you have non-default values for `/vis/scene/endOfEventAction` or `/vis/scene/endOfRunAction` (described below).

**Command:** `/vis/scene/add/trajectories [smooth] [rich]`

- **Action**
  The command adds trajectories to the current scene. Trajectories are drawn at end of event when the scene in which they are added is current.
- **Example: Visualization of trajectories**

```
/vis/scene/add/trajectories
/run/beamOn 10
```

- **Additional note 1**
  See the section *Controlling from Commands* for details on how trajectories are color-coded.
- **Additional note 2**
  Events may be kept and reviewed at end of run with:

```
/vis/reviewKeptEvents
```

  Keep all events with:

```
/vis/scene/endOfEventAction accumulate [maxNumber]
```

  (see *End of Event Action and End of Run Action: /vis/scene/endOfEventAction and /vis/scene/endOfRunAction commands*)
  or keep some chosen subset by some selection in your user code, for example your user event action:

```
if ( some criterion ) {
  G4EventManager::GetEventManager()->KeepTheCurrentEvent();
}
```

  or:

```
if ( some criterion ) {
  UImanager->ApplyCommand("/event/keepCurrentEvent");
}
```

as described in Listing 6.8.
To draw only those events kept as above:

```
/vis/drawOnlyToBeKeptEvents
```

To suppress drawing during a run:

```
/vis/disable
/run/beamOn 10000
```

then at end of run:

```
/vis/enable
/vis/reviewKeptEvents
```

- **Additional note 3**
  Visualising events as they are being generated inevitably slows the simulation. Visualisation can be suspended with `/vis/disable` as suggested above. You may also switch off trajectory production with `/tracking/storeTrajectory 0`. When using OpenGL, the following can help:

  ```
  /vis/ogl/flushAt
  <[endOfEvent|endOfRun|eachPrimitive|NthPrimitive|NthEvent|never]> <N>
  ```

  By default, this value is set to `/vis/ogl/flushAt NthEvent 100`

For more options, see the Control…UICommands section of this user guide.

## 8.4.8 Visualization of hits: `/vis/scene/add/hits` command

Command "`/vis/scene/add/hits`" adds hits to the current scene, assuming that you have a hit class and that the hits have visualization information. The visualization is performed with the command "`/run/beamOn`" unless you have non-default values for /vis/scene/endOfEventAction or /vis/scene/endOfRunAction (described above).

## 8.4.9 Visualization of fields: `/vis/scene/add/magneticField` command

`/vis/scene/add/magneticField` and `/vis/scene/add/electricField` will draw any fields defined in the scene as an array of arrows whose direction, length and colour are related to the field strength and direction.

Sometimes this can result in a overwhelming number of arrows. To limit the extent of the arrows preface one or more of the above commands with:

```
/vis/set/extentForField
```

or:

```
/vis/set/volumeForField
```

or equivalent commands in `/vis/touchable/`.

This can be repeated to get the desired effect, e.g.:

```
/vis/set/extentForField -20 20 -55 0 0 50 cm
/vis/scene/add/magneticField
/vis/set/volumeForField detector1
/vis/scene/add/magneticField
/vis/set/volumeForField detector2 5
/vis/scene/add/electricField
```

To remove fields from the scene:

```
/vis/scene/activateModel Field false
```

Consult the guidance for the `/vis/scene/add/...Field` commands for further hints and suggestions.

### 8.4.10 Visualization of Scored Data

Scored data can be visualized using the commands "`/score/drawProjection`" and "`/score/drawColumn`". For details, see examples/extended/runAndEvent/RE03.

### 8.4.11 Additional attributes for Hits

The HepRep file formats, HepRepFile and HepRepXML, understand various additional attributes such that you can view these attributes, label trajectories by these attributes or make visibility cuts based on these attributes. Examples of adding HepRep attributes to hit classes can be found in examples /extended/analysis/A01 and /extended/runAndEvent/RE01.

For example, in example RE01's class RE01CalorimeterHit.cc, available attributes will be:

- Hit Type
- Track ID
- Z Cell ID
- Phi Cell ID
- Energy Deposited
- Energy Deposited by Track
- Position
- Logical Volume

You can add additional attributes of your choosing by modifying the relevant part of the hit class (look for the methods GetAttDefs and CreateAttValues).

### 8.4.12 Visualization of histograms (plotting)

The G4/vis system is equipped to be able to do plotting, then to have a representation (a plot) of 1D or 2D histograms within a G4/vis viewer. The G4 vis primitive G4Plotter has been introduced to capture which histograms to plot, specify a grid of plots (2x2, 2x3, etc…), along some style options to customize the representations (for example to change bins color, title or axis label fonts, etc…). Specifying a grid of plots (or "regions") is a common practice in plotting and is a similar concept as the "zones" found in the good old CERN/PAW.

The known histograms are the ones managed in G4/analysis and are known in G4Plotter by using their integer id.

This said, each specific vis driver is charged with implementing the representation of a G4Plotter. Today only the ToolsSG drivers come with such representation, but we hope that more vis drivers will come with an implementation in the future.

From the user point of view, commands has been introduced to be able to specify a plot from pure .mac scripting. To start with, the best is to jump in examples/basic/B5 that comes with a commented plotter.mac example. In it you will see how to activate the vis driver (create a "scene handler"), create a viewer, create a scene containing a plotter model object (then a G4Plotter), create a grid of plotting "regions" (here 2x2 regions) and attach the histograms to each region. When done, each "run beamOn" should display at end the content of the histograms.

The skeleton of a plotting script then looks like:

```
# viewer:
/vis/sceneHandler/create TSG scene-handler-plotter
/vis/viewer/create scene-handler-plotter viewer-plotter 600x600-0+0
/vis/viewer/set/background 1 1 1
/vis/viewer/zoomTo 1
/vis/viewer/set/viewpointVector 0 0 1
# scene:
/vis/plotter/create plotter-0
/vis/scene/create scene-plotter
/vis/scene/add/plotter plotter-0
/vis/sceneHandler/attach scene-plotter
# create a 2x2 plotter regions:
/vis/plotter/setLayout plotter-0 2 2
# attach histograms to regions (examples/basic/B5 specific):
/vis/plotter/add/h1 0 plotter-0 0
/vis/plotter/add/h1 1 plotter-0 1
/vis/plotter/add/h2 0 plotter-0 2
/vis/plotter/add/h2 1 plotter-0 3
# let's go:
/run/beamOn 100
# the upper will update the plotters at end of run.
```

**Plotting style:**

Being able to customize the representations is an important part of plotting. The concept of named style had been introduced in G4/vis to handle this. A named style is nothing more than a named list of pairs (key,value) which is managed in the G4/vis system.

You can create a named style and set it as “current style” with:

```
/vis/plotting/style/select <name>
```

You can deposit pairs of key/value in the current style with:

```
/vis/plotter/style/add <key> <value>
/vis/plotter/style/add <key> <value>
...
```

A named style can be used on a specific region with:

```
/vis/plotter/addRegionStyle <plotter> <region> <style>
```

For example, in B5/plotter.mac:

```
/vis/plotter/addRegionStyle plotter-0 0 style-0
```

or on a whole grid of plots with:

```
/vis/plotter/addStyle <plotter> <style>
```

Note that someone can add multiple named styles on a plotter or on a region. If so the styles are applied in order with global ones first and then per region ones after.

IMPORTANT: the key/value pairs are specific of a G4Plotter representation implementation. If using the ToolsSG plotting you may have:

```
/vis/plotter/style/add bins_style.0.color blue
/vis/plotter/style/add bins_style.0.line_width 3
/vis/plotter/style/add infos_width 0.2
/vis/plotter/style/add infos_style.visible true
/vis/plotter/style/add infos_style.font roboto_bold.ttf
/vis/plotter/style/add infos_style.front_face cw
```

but, a priori, the upper key/value pairs are not expected to be known by another plotter implementation. (But it would be great to be so!).

Other usefull style commands are:

```
# to list known named styles.
/vis/plotter/style/list
# to print the list of key/value pairs of a named style:
/vis/plotter/style/print <style>
# to remove a name style from G4/vis:
/vis/plotter/style/remove <style>
```

Note that without passing by a style, a plotting region can be customised directly by using the command:

```
/vis/plotter/addRegionParameter <plotter> <region> <key> <value>
```

for example with ToolsSG plotting in B5/plotter.mac:

```
/vis/plotter/addRegionParameter plotter-0 0 bins_style.0.color blue
/vis/plotter/addRegionParameter plotter-0 0 bins_style.0.line_width 3
```

As for styles, the key/value pairs are specific of a vis driver plotting implementation. Note that the pairs given with the addRegionParameter on a region are applied after all styles on this region.

**ToolsSG plotting:**

The ToolsSG/G4Plotter representation is done by using the high level tools::sg::plots and tools::sg::plotter nodes (found in externals/g4tools/include/tools/sg). The tools::sg::plotter node uses a lot of nodes as tools::sg::axis, tools::sg::vertices, etc.. to build representations. (tools::sg::plots permits to implement a grid of plots). A tools::sg::node manages "smart fields", for example the tools::sg::sf<float> (simple float smart field) "width" in tools::sg::plotter. (Smart field is a similar concept as the SoField in OpenInventor). It is these fields that are customizable as key/value pairs from styles or region parameters. (A smart field is smart in the sense that if "touched", for example by setting a new value, it may induce an automatic update of the node at next rendering traversal of a scene graph).

The list of what is customizable on a tools::sg::plotter, is given with the ToolsSG specific command:

```
/vis/tsg/plotter/printParameters
```

For styles, specific to ToolsSG plotting, are the named styles "default", "ROOT_default", "hippodraw" that are "embedded" styles (defined as C++ functions in: tools/sg/plotter_some_styles). (These styles are the same than in G4/analysis for batch plotting). In particular the ROOT_default styles permits to mimic the default style plotting found in CERN/ROOT. Then someone can do:

```
/vis/plotter/addStyle <plotter> ROOT_default
```

to have the ROOT style for all regions. (Note that ROOT_default needs freetype).

In the second part of B5 plotter.mac, is shown various ways to customize the regions, for example changing the bins color, the axis labels fonts, etc… This could be done by using default embedded styles, defining styles with commands, or setting up directly parameters of the various parts of a plot by using the dedicated addRegionParameter command.

For texts (title, tick labels, etc…), the fonts used by default are the Hershey vectorial ones (the ones of the good old CERN/PAW) that do not need an extra package, but you can use some freetype fonts if building with the cmake flag -DGEANT4_USE_FREETYPE=ON. For example the ROOT_default embedded style uses freetype fonts. Two embedded ttf fonts come with the ToolsSG plotting: roboto_bold (some open source kind of the Microsoft arialbd) and lato_regular (close to an helvetica). You can use also your own .ttf files by using the TOOLS_FONT_PATH environment variable to specify the directory where they could be found.

**ToolsSG plotting keys:**

A style or parameter key has the form:

```
<direct field> of sg::plotter, as width, height, left_margin, right_margin, etc...
```

or is a field of a sub node representing a component of the scene of the plot, such as [x,y,z] axis, infos box, title box, grid, bins, errors, etc… The key may refer a direct field of the component such as x_axis.title, or the style of a component handled by a tools::sg::style or sg::text_style node. A component key may contain two or three words separated by a dot.

Two words keys are for:

```
[x_axis,y_axis,z_axis].<field>
```

For example:

```
x_axis.modeling   (string field with value hplot, hippodraw).
x_axis.divisions  (in case of hplot modeling, an int specifiying primary/secondary ticks encoded␣
↪as in hplot (for exa 510)).
```

or for the style components:

```
background_style, title_style, infos_style, title_box_style,
inner_frame_style, grid_style, wall_style
```

For example:

```
infos_style.visible
infos_style.font
infos_style.front_face
```

Three words keys are also for the style of the components of the tools::sg::axis as:

```
line_style, ticks_style, labels_style, mag_style, title_style
```

for example:

```
x_axis.labels_style.color
```

Three words keys are used also to specified style fields of the bins, errors, function, points, legend data representations components. For example, in one sg::plotter, you can specify to plot multiple histograms, ie multiple "bins". In this case, you can customize the style of the i-th "bins" (i-th histogram) with a key of the form:

```
bins_style.<i-th>.<field>
```

for example to change color of the "front" histogram:

```
bins_style.0.color
```

For the moment, the G4/vis plotting knows only histograms, but the tools::sg::plotter can handle cloud of points, errors, functions, legends, and in some future, a G4 user may have to use the errors_style, func_style, points_style, legend_style in the same way to customize a "i-th" cloud of points, a i-th function, a i-th legend, etc…

We do not give here the full list of available parameters since it may evolve in time. The best is to use the command:

```
/vis/tsg/plotter/printParameters
```

that dumps, by querying directly the nodes, the available styles and smart fields for the sg::plotter itself or for one of its component. Moreover it dumps also the type of a field (float, integer, boolean, string, etc…).

Put all together, the combinatory of available keys is rather rich and permits a strong customization of good parts of a sg::plotter.

### 8.4.13 Basic camera workings: `/vis/viewer/` commands

Commands in the command directory "`/vis/viewer/`" set camera parameters and drawing style of the current viewer, which corresponds to Step 5. Note that the camera parameters and the drawing style should be set separately for each viewer. They can be initialized to the default values with command "`/vis/viewer/reset`". Some visualization systems, such as the VRML and HepRep browsers also allow camera control from the standalone graphics application.

Just a few of the camera commands are described here. For more commands, see the Control…UICommands section of this user guide.

The view is defined by a target point (initially at the centre of the extent of all objects in the scene), an up-vector and a viewpoint direction - see Fig. 8.1. By default, the up-Vector is parallel to the y-axis and the viewpoint direction is parallel to the z-axis, so the the view shows the x-axis to the right and the y-axis upwards - a projection on to the canonical x-y plane - see Fig. 8.2 figure.

The target point can be changed with a `/vis/viewer/set` command or with the `/vis/viewer/pan` commands. The up-vector and the viewpoint direction can also be changed with `/vis/viewer/set` commands. Care must be taken to avoid having the two vectors parallel, for in that case the view is undefined.

The commands:

```
/vis/viewer/centreOn <volume-name> [<copy-number>]
/vis/viewer/centreAndZoomInOn <volume-name> [<copy-number>]
```

also change the target point.

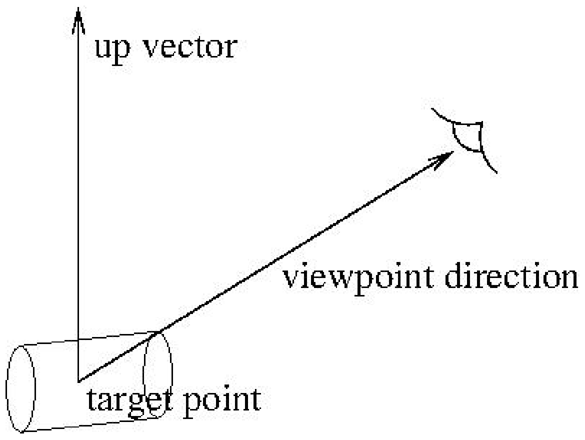


Fig. 8.1: Up-vector and viewpoint direction

**Command:** `/vis/viewer/set/viewpointThetaPhi [theta] [phi] [deg|rad]`

- **Arguments**
  Arguments "theta" and "phi" are polar and azimuthal camera angles, respectively. The default unit is "degree".
- **Action**
  Set a view point in direction of (theta, phi).
- **Example: Set the viewpoint in direction of (70 deg, 20 deg) /**

```
/vis/viewer/set/viewpointThetaPhi 70 20
```

- **Additional notes**
  Camera parameters should be set for each viewer. They are initialized with command "`/vis/viewer/reset`". Alternatively, they can be copied from another viewer with the command "`/vis/viewer/copyViewFrom viewer-0`", for example.

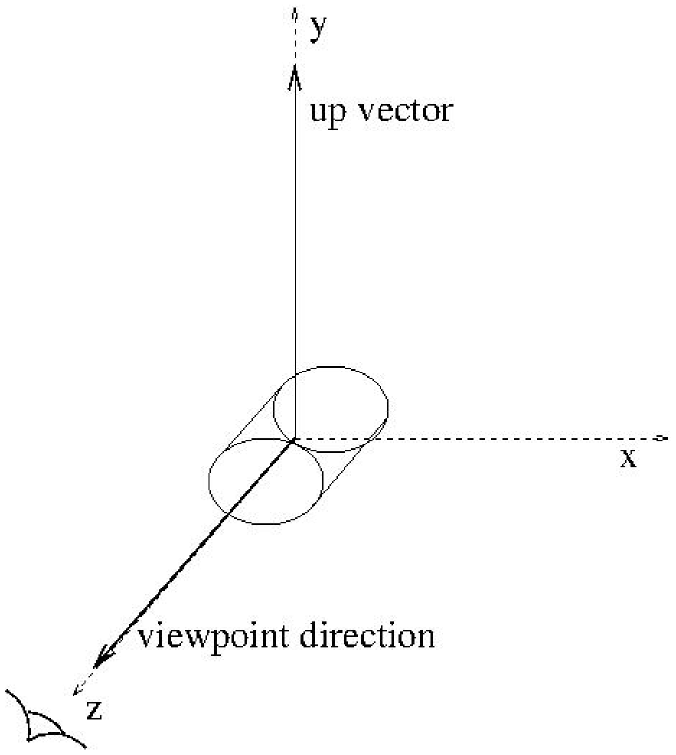


Fig. 8.2: The default view

**Command:** `/vis/viewer/zoom [scale_factor]`

- **Argument**
  The scale factor. The command multiplies magnification of the view by this factor.
- **Action**
  Zoom up/down of view.
- **Example: Zoom up by factor 1.5**

```
/vis/viewer/zoom 1.5
```

- **Additional notes**
  A similar pair of commands, scale and scaleTo allow non-uniform scaling (i.e., zoom differently along different axes). For details of this and lots of other commands, see the Control…UICommands section of this user guide. Some viewers have limits to how large the zoom factor can be. This problem can be circumnavigated to some degree by using `zoom` and `scale` together. If

```
/vis/viewer/zoomTo 1e10
```

  does not work, please try

```
/vis/viewer/scaleTo 1e5 1e5 1e5
/vis/viewer/zoomTo 1e5
```

  Of course, with such high zoom factors, you might want to know whither you are zooming. Use `/vis/viewer/set/targetPoint` or `/vis/viewer/centreOn` or `/vis/viewer/centreAndZoomInOn`.
  Camera parameters should be set for each viewer. They are initialized with command "`/vis/viewer/reset`". Alternatively, they can be copied from another viewer with the command "`/vis/viewer/copyViewFrom viewer-0`", for example.

**Command:** `/vis/viewer/set/style [style_name]`

- **Arguments**
  Candidate values of the argument are "wireframe" and "surface". ("w" and "s" also work.)
- **Action**
  Set a drawing style to wireframe or surface.
- **Example: Set the drawing style to "surface"**

```
/vis/viewer/set/style surface
```

- **Additional notes**
  The style of some geometry components may have been forced one way or the other through calls in compiled code. The set/style command will NOT override such force styles.
  Drawing style should be set for each viewer. The drawing style is initialized with command "`/vis/viewer/reset`". Alternatively, it can be copied from another viewer with the command "`/vis/viewer/set/all viewer-0`", for example.

### 8.4.14 Declare the end of visualization for flushing: `/vis/viewer/flush` command

**Command:** `/vis/viewer/flush`

- **Action**
  Declare the end of visualization for flushing.
- **Additional notes**
  Command "`/vis/viewer/flush`" should follow "`/vis/drawVolume`", "`/vis/specify`", etc in order to complete visualization. It corresponds to Step 7.
  The flush is done automatically after every /run/beamOn command unless you have non-default values for /vis/scene/endOfEventAction or /vis/scene/endOfRunAction (described above).

### 8.4.15 End of Event Action and End of Run Action: `/vis/scene/endOfEventAction` and `/vis/scene/endOfRunAction` commands

By default, a separate picture is created for each event. You can change this behaviour to accumulate multiple events, or even multiple runs, in a single picture.

**Command:** `/vis/scene/endOfEventAction [refresh|accumulate]`

- **Action**
  Control how often the picture should be cleared. `refresh` means each event will be written to a new picture. `accumulate` means events will be accumulated into a single picture. Picture will be flushed at end of run, unless you have also set `/vis/scene/endOfRunAction accumulate`
- **Additional note**
  You may instead choose to use update commands from your BeginOfRunAction or EndOfEventAction, as in early examples, but now the vis manager is able to do most of what most users require through the above commands.

**Command:** `/vis/scene/endOfRunAction [refresh|accumulate]`

- **Action**
  Control how often the picture should be cleared. `refresh` means each run will be written to a new picture. `accumulate` means runs will be accumulated into a single picture. To start a new picture, you must explicitly issue `/vis/viewer/refresh`, `/vis/viewer/update` or `/vis/viewer/flush`

### 8.4.16 HepRep Attributes for Trajectories

The HepRep file formats, HepRepFile and HepRepXML, attach various attributes to trajectories such that you can view these attributes, label trajectories by these attributes or make visibility cuts based on these attributes. If you use the default GEANT4 trajectory class from /tracking/src/G4Trajectory.cc (this is what you get with the plain `/vis/scene/add/trajectories` command), available attributes will be:

- Track ID
- Parent ID
- Particle Name
- Charge
- PDG Encoding
- Momentum 3-Vector
- Momentum magnitude
- Number of points

Using `/vis/scene/add/trajectories rich` will get you additional attributes. You may also add additional attributes of your choosing by modifying the relevant part of G4Trajectory (look for the methods GetAttDefs and CreateAttValues). If you are using your own trajectory class, you may want to consider copying these methods from G4Trajectory.

### 8.4.17 How to save a view.

```
/vis/viewer/save
```

This will save to a file that can be read in again with

```
/control/execute
```

If you save several views you may "fly through" them with

```
/vis/viewer/interpolate
```

See *Making a Movie*.

(Use the GEANT4 "help" command to see details.)

### 8.4.18 How to save a view to an image file

Most of the visualization drivers offer ways to save visualized views to PostScript (PS) or Encapsulated PostScript (EPS). Some, in addition, offer Portable Document Format (PDF). OpenGL offers a big range of formats - see below.

- **DAWNFILE**
  The DAWNFILE driver, which co-works with Fukui Renderer DAWN, generates "vectorized" PostScript data with "analytical hidden-line/surface removal", and so it is well suited for technical high-quality outputs for presentation, documentation, and debugging geometry. In the default setting of the DAWNFILE drivers, EPS files named "`g4_00.eps, g4_01.eps, g4_02.eps,...`" are automatically generated in the current directory each time when visualization is performed, and then a PostScript viewer "`gv`"is automatically invoked to visualize the generated EPS files.
  For large data sets, it may take time to generate the vectorized PostScript data. In such a case, visualize the 3D scene with a faster visualization driver beforehand for previewing, and then use the DAWNFILE drivers. For example, the following visualizes the whole detector with the OpenGL-Xlib driver (immediate mode) first, and then with the DAWNFILE driver to generate an EPS file `g4_XX.eps` to save the visualized view:

```
# Invoke the OpenGL visualization driver in its immediate mode
/vis/open OGLIX

# Camera setting
/vis/viewer/set/viewpointThetaPhi 20 20

# Camera setting
/vis/drawVolume
/vis/viewer/flush

# Invoke the DAWNFILE visualization driver
/vis/open DAWNFILE

# Camera setting
/vis/viewer/set/viewpointThetaPhi 20 20

# Camera setting
/vis/drawVolume
/vis/viewer/flush
```

This is a good example to show that the visualization drivers are complementary to each other.

- **OpenInventor**
  In the OpenInventor drivers, you can simply click the "Print" button on their GUI to generate a PostScript file as a hard copy of a visualized view.
- **OpenGL**
  The OpenGL drivers can also generate image files, either from a pull-down menu (Motif and Qt drivers) or with `/vis/ogl/export`. Available formats are: eps ps pdf svg bmp cur dds icns ico jp2 jpeg jpg pbm pgm png ppm tif tiff wbmp webp xbm xpm. The default is pdf. It can generate either vector or bitmap PostScript data with `/vis/ogl/set/printMode` ("vectored" or "pixmap"). You can change the filename by `/vis/ogl/set/printFilename` And the print size by `/vis/ogl/set/printSize` In generating vectorized PostScript data, hidden-surface removal is performed based on the painter's algorithm after dividing facets of shapes into small sub-triangles.
  The `/vis/ogl/set/printSize` command can be used to print EPS files even larger than the current screen resolution. This can allow creation of very large images, suitable for creation of posters, etc. The only size limitation is the graphics card's viewport dimension: GL_MAX_VIEWPORT_DIMS

```
# Invoke the OpenGL visualization driver in its stored mode
/vis/open OGLSX

# Camera setting
/vis/viewer/set/viewpointThetaPhi 20 20

# Camera setting
/vis/drawVolume
/vis/viewer/flush

# set print mode to vectored
#/vis/ogl/set/printMode vectored

# set print size larger than screen
/vis/ogl/set/printSize 2000 2000

# print
/vis/ogl/export
```

- **HepRep**
  The HepRApp HepRep Browser such as FRED can generate a wide variety of bitmap and vector output formats including PostScript and PDF.

## 8.4.19 Culling

"Culling" means to skip visualizing parts of a 3D scene. Culling is useful for avoiding complexity of visualized views, keeping transparent features of the 3D scene, and for quick visualization.

GEANT4 Visualization supports the following 3 kinds of culling:

- Culling of invisible physical volumes
- Culling of low density physical volumes.
- Culling of covered physical volumes by others

In order that one or all types of the above culling are on, i.e., activated, the global culling flag should also be on.

Table 8.3 summarizes the default culling policies.

Table 8.3: The default culling policies.

| Culling Type | Default Value |
|---|---|
| global | ON |
| invisible | ON |
| low density | OFF |
| covered daughter | OFF |

The default threshold density of the low-density culling is 0.01 g/cm$^3$.

The default culling policies can be modified with the following visualization commands. (Below the argument `flag` takes a value of `true` or `false`.)

```
# global
/vis/viewer/set/culling  global  flag

# invisible
/vis/viewer/set/culling  invisible  flag

# low density
#    "value" is a proper value of a threshold density
#    "unit" is either g/cm3, mg/cm3 or kg/m3
/vis/viewer/set/culling  density  flag  value  unit

# covered daughter
/vis/viewer/set/culling  coveredDaughters  flag     density
```

The HepRepFile graphic system will, by default, include culled objects in the file so that they can still be made visible later from controls in the HepRep browser. If this behavior would cause files to be too large, you can instead choose to have culled objects be omitted from the HepRep file. See details in the HepRepFile Driver section of this user guide.

## 8.4.20 Cut view

### Sectioning

"Sectioning" means to make a thin slice of a 3D scene around a given plane. At present, this function is supported by the OpenGL drivers. The sectioning is realized by setting a sectioning plane before performing visualization. The sectioning plane can be set by the command,

```
/vis/viewer/set/sectionPlane on x y z units nx ny nz
```

where the vector (x,y,z) defines a point on the sectioning plane, and the vector (nx,ny,nz) defines the normal vector of the sectioning plane. For example, the following sets a sectioning plane to a yz plane at x = 2 cm:

```
/vis/viewer/set/sectionPlane on 2.0 0.0 0.0 cm 1.0 0.0 0.0
```

**Cutting away**

"Cutting away" means to removing a half space from a 3D scene. It is available for all drivers. (The OpenGL driver has its own implementation that uses OpenGL cut planes. DAWNFILE has a special way - see below. Other drivers use a "generic" algorithm based on Boolean subtractions and/or intersections.)

- Add up to three cutaway planes:

  ```
  /vis/viewer/addCutawayPlane 0 0 0 m 1 0 0
  /vis/viewer/addCutawayPlane 0 0 0 m 0 1 0
  ...
  ```

  and, for more that one plane, you can change the mode to
  - "add" or, equivalently, "union" (default) or
  - "multiply" or, equivalently, "intersection":

    ```
    /vis/viewer/set/cutawayMode multiply
    ```

  To de-activate:

  ```
  /vis/viewer/clearCutawayPlanes
  ```

- Cutting is supported by the DAWNFILE driver "off-line". Do the following:
  - Perform visualization with the DAWNFILE driver to generate a file `g4.prim`, describing the whole 3D scene.
  - Make the application "DAWNCUT" read the generated file to make a view of cutting away.

### 8.4.21 Multithreading commands

Visualising events inevitably slows things down. With multithreading this effect is all the greater. See *Visualization of trajectories: /vis/scene/add/trajectories command*, Additional Note 3, for some advice. If you wish to continue visualising, multithreading mode offers the following fine tuning.

Since GEANT4 version 10.2, in multithreading mode, events generated by worker threads are put in a queue and extracted by a special visualisation thread. If the queue gets full, workers are suspended until the visualisation thread catches up. To mitigate or avoid this try using

```
/vis/multithreading/maxEventQueueSize <N>
/vis/multithreading/actionOnEventQueueFull <wait|discard>
```

(See command guidance for details.)

## 8.5 Controlling Visualization from Compiled Code

While a GEANT4 simulation is running, visualization can be performed without user intervention. This is accomplished by calling methods of the Visualization Manager from methods of the user action classes (`G4UserRunAction` and `G4UserEventAction`, for example). In this section methods of the class `G4VVisManager`, which is part of the `graphics_reps` category, are described and examples of their use are given.

### 8.5.1 G4VVisManager

The Visualization Manager is implemented by classes `G4VisManager` and `G4VisExecutive`. See *Adding Visualization to Your Executable*. In order that your GEANT4 be compilable either with or without the visualization category, you should not use these classes directly in your C++ source code, other than in the `main()` function. Instead, you should use their abstract base class `G4VVisManager`, defined in the `intercoms` category.

The pointer to the concrete instance of the real Visualization Manager can be obtained as follows:

```
//----- Getting a pointer to the concrete Visualization Manager instance
G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();
```

The method `G4VVisManager::GetConcreteInstance()` returns `NULL` if GEANT4 is not ready for visualization. Thus your C++ source code should be protected as follows:

```
//----- How to protect your C++ source codes in visualization
if (pVVisManager) {
    ....
    pVVisManager ->Draw (...);
    ....
}
```

Note: It pays to encapsulate your `Draw` messages in *Visualization User Actions*. The vis manager then has control over the drawing and may call your action as required, for example, to refresh the screen or write to file.

### 8.5.2 Visualization of detector components

If you have already constructed detector components with logical volumes to which visualization attributes are properly assigned, you are almost ready for visualizing detector components. The usual and *recommended* way is to use UI commands - see *Controlling Visualization from Commands*.

Most examples have a file `vis.mac` that is executed by default in interactive mode.

However, if you really wish to program visualisation we recommended simply using the `ApplyCommand()` method as below:

```
//----- C++ source code: How to visualize detector components (2)
//                        ... using visualization commands in source codes

G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance() ;

if(pVVisManager)
{
    ... (camera setting etc) ...
    G4UImanager::GetUIpointer()->ApplyCommand("/vis/drawVolume");
    G4UImanager::GetUIpointer()->ApplyCommand("/vis/viewer/flush");
}

//-----  end of C++ source code
```

For the adventurous user, the vis manager offers methods such as:

```
virtual void Draw (const G4VPhysicalVolume&, const G4VisAttributes&,
  const G4Transform3D& objectTransformation = G4Transform3D()) = 0;

virtual void DrawGeometry
(G4VPhysicalVolume*, const G4Transform3D& t = G4Transform3D());
// Draws a geometry tree starting at the specified physical volume.
```

### 8.5.3 Visualization of trajectories

Again, we *recommend* using commands - see /vis/modeling and vis/filtering.

But you may specialise by writing C++ code, for example in void G4Trajectory::DrawTrajectory() defined in the tracking category. The vis manager offers a good collection of Draw methods. For example:

```
//----- A drawing method of G4Polyline
virtual void G4VVisManager::Draw (const G4Polyline&, ...) ;
```

Your DrawTrajectory will then be used by the vis manager when you add trajectories to the scene - see *Visualization of trajectories: /vis/scene/add/trajectories command*.

Alternatively, you may pick up trajectories from a G4TrajectoryContainer at end of event and invoke your DrawTrajectory:

```
void ExN03EventAction::EndOfEventAction(const G4Event* evt)
{
  .....
  // extract the trajectories and draw them
  if (G4VVisManager::GetConcreteInstance())
    {
     G4TrajectoryContainer* trajectoryContainer = evt->GetTrajectoryContainer();
     G4int n_trajectories = 0;
     if (trajectoryContainer) n_trajectories = trajectoryContainer->entries();

     for (G4int i=0; i < n_trajectories; i++)
        { G4Trajectory* trj=(G4Trajectory*)((*(evt->GetTrajectoryContainer()))[i]);
          if (drawFlag == "all") trj->DrawTrajectory(50);
          else if ((drawFlag == "charged")&&(trj->GetCharge() != 0.))
                                  trj->DrawTrajectory(50);
          else if ((drawFlag == "neutral")&&(trj->GetCharge() == 0.))
                                  trj->DrawTrajectory(50);
        }
  }
}
```

### 8.5.4 Enhanced trajectory drawing

It is possible to use the enhanced trajectory drawing functionality in compiled code as well as from commands. Multiple trajectory models can be instantiated, configured and registered with G4VisManager. For details, see the section on *Controlling from Compiled Code*.

### 8.5.5 HepRep Attributes for Trajectories

The HepRep file format, HepRepFile, attaches various attributes to trajectories such that you can view these attributes, label trajectories by these attributes or make visibility cuts based on these attributes. If you use the default GEANT4 trajectory class, from /tracking/src/G4Trajectory.cc (which is what you get with /vis/scene/add/trajectories) the available attributes will be:

- Track ID
- Parent ID
- Particle Name
- Charge
- PDG Encoding
- Momentum 3-Vector
- Momentum magnitude
- Number of points

A more extensive list of attributes is available with `G4RichTrajectory` (`/vis/scene/add/trajectories rich`).

You can add additional attributes of your choosing by modifying the relevant part of G4[Rich]Trajectory (look for the methods GetAttDefs and CreateAttValues). If you are using your own trajectory class, you may want to consider copying these methods from G4Trajectory.

## 8.5.6 Visualization of hits

There is no default code for drawing hits. You have to write a `Draw()` method in your hit class. Similarly `DrawAllHits()` in your hits collection class. You can use drawing methods of class `G4VVisManager`:

```
virtual void G4VVisManager::Draw (const G4Circle&, ...);
virtual void G4VVisManager::Draw (const G4Square&, ...);
virtual void G4VVisManager::Draw (const G4VPhysicalVolume&, ...);
...
```

For example, class *MyTrackerHits* inheriting `G4VHit`:

```
//----- An example of giving concrete implementation of
//       G4VHit::Draw(), using  class MyTrackerHit : public G4VHit {...}
//
void MyTrackerHit::Draw()
{
   G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();
   if(pVVisManager)
   {
     // define a circle in a 3D space
     G4Circle circle(pos);
     circle.SetScreenSize(0.3);
     circle.SetFillStyle(G4Circle::filled);

     // make the circle red
     G4Colour colour(1.,0.,0.);
     G4VisAttributes attribs(colour);
     circle.SetVisAttributes(attribs);

     // make a 3D data for visualization
     pVVisManager->Draw(circle);
   }
 }
```

Your `DrawAllHits()` method could be:

```
//----- An example of giving concrete implementation of
//       G4VHitsCollection::Draw(),
//       using  class MyTrackerHit : public G4VHitsCollection{...}
//
void MyTrackerHitsCollection::DrawAllHits()
{
  G4int n_hit = theCollection.entries();
  for(G4int i=0;i < n_hit;i++)
  {
    theCollection[i].Draw();
  }
}
```

The *recommended* way to invoke these functions is to add hits to the scene:

```
/vis/scene/add/hits
```

In this case the vis manager will invoke them as required.

Alternatively, as with trajectories, you may, if you wish, draw from your `EndOfEventAction`:

```
void MyEventAction::EndOfEventAction()
{
  const G4Event* evt = fpEventManager->GetConstCurrentEvent();

  G4SDManager * SDman = G4SDManager::GetSDMpointer();
  G4String colNam;
  G4int trackerCollID = SDman->GetCollectionID(colNam="TrackerCollection");
  G4int calorimeterCollID = SDman->GetCollectionID(colNam="CalCollection");

  G4TrajectoryContainer * trajectoryContainer = evt->GetTrajectoryContainer();
  G4int n_trajectories = 0;
  if(trajectoryContainer)
  { n_trajectories = trajectoryContainer->entries(); }

  G4HCofThisEvent * HCE = evt->GetHCofThisEvent();
  G4int n_hitCollection = 0;
  if(HCE)
  { n_hitCollection = HCE->GetCapacity(); }

  G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();

  if(pVVisManager) {

    // Declare begininng of visualization
    G4UImanager::GetUIpointer()->ApplyCommand("/vis/scene/notifyHandlers");

    // Draw trajectories
    for(G4int i=0; i < n_trajectories; i++) {
        (*(evt->GetTrajectoryContainer()))[i]->DrawTrajectory();
    }

    // Construct 3D data for hits
    MyTrackerHitsCollection* THC
      = (MyTrackerHitsCollection*)(HCE->GetHC(trackerCollID));
    if(THC) THC->DrawAllHits();
    MyCalorimeterHitsCollection* CHC
      = (MyCalorimeterHitsCollection*)(HCE->GetHC(calorimeterCollID));
    if(CHC) CHC->DrawAllHits();

    // Declare end of visualization
    G4UImanager::GetUIpointer()->ApplyCommand("/vis/viewer/update");
  }
}
```

You can re-visualize a physical volume, where a hit is detected, with a highlight color, in addition to the whole set of detector components. It is done by calling `G4VVisManager::Draw(const G4VPhysicalVolume&, ...)`:

```
//----- An example of visualizing hits with a physical volume
void MyCalorimeterHit::Draw()
{
  G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();
  if(pVVisManager)
  {
    G4Transform3D trans(rot,pos);
    G4VisAttributes attribs;
    G4LogicalVolume* logVol = pPhys->GetLogicalVolume();
    const G4VisAttributes* pVA = logVol->GetVisAttributes();
    if(pVA) attribs = *pVA;
    G4Colour colour(1.,0.,0.);
    attribs.SetColour(colour);
    attribs.SetForceSolid(true);

    //----- Re-visualization of a selected physical volume with red color
```

(continues on next page)

(continued from previous page)

```
    pVVisManager->Draw(*pPhys,attribs,trans);
  }
}
```

### 8.5.7 HepRep Attributes for Hits

The HepRep file format, HepRepFile, attaches various attributes to hits such that you can view these attributes, label trajectories by these attributes or make visibility cuts based on these attributes. Examples of adding HepRep attributes to hit classes can be found in examples /extended/analysis/A01 and /extended/runAndEvent/RE01.

For example, in example RE01's class RE01CalorimeterHit.cc, available attributes will be:

- Hit Type
- Track ID
- Z Cell ID
- Phi Cell ID
- Energy Deposited
- Energy Deposited by Track
- Position
- Logical Volume

You can add additional attributes of your choosing by modifying the relevant part of the hit class (look for the methods GetAttDefs and CreateAttValues).

### 8.5.8 Visualization of text

In GEANT4 Visualization, a text, i.e., a character string, is described by class `G4Text` inheriting `G4VMarker` as well as `G4Square` and `G4Circle`. Therefore, the way to visualize text is the same as for hits. The corresponding drawing method of `G4VVisManager` is:

```
//----- Drawing methods of G4Text
virtual void G4VVisManager::Draw (const G4Text&, ...);
```

The real implementation of this method is described in class `G4VisManager`.

### 8.5.9 Visualization of polylines and tracking steps

We remind the reader that the vis manager provides a generous selection of UI commands to draw and filter trajectories and thus to see the tracking steps - see *Visualization of trajectories: /vis/scene/add/trajectories command*.

Alternatively, if you wish, you may code your own functions. Polylines, i.e., sets of successive line segments, are described by class `G4Polyline`. For `G4Polyline`, the following drawing method of class `G4VVisManager` is prepared:

```
//----- A drawing method of G4Polyline
 virtual void G4VVisManager::Draw (const G4Polyline&, ...) ;
```

The real implementation of this method is described in class `G4VisManager`.

Using this method, C++ source codes to visualize `G4Polyline` are described as follows:

```
//----- How to visualize a polyline
 G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();

 if (pVVisManager) {
     G4Polyline polyline ;

     ..... (C++ source codes to set vertex positions, color, etc)

     pVVisManager -> Draw(polyline);
 }
```

Tracking steps are able to be visualized based on the above visualization of `G4Polyline`. You can visualize tracking steps at each step automatically by writing a proper implementation of class *MySteppingAction* inheriting `G4UserSteppingAction`, and also with the help of the Run Manager.

First, you must implement a method, `MySteppingAction::UserSteppingAction()`. A typical implementation of this method is as follows:

```
//-----  An example of visualizing tracking steps
void MySteppingAction::UserSteppingAction()
{
    G4VVisManager* pVVisManager = G4VVisManager::GetConcreteInstance();

    if (pVVisManager) {
      //----- Get the Stepping Manager
      const G4SteppingManager* pSM = GetSteppingManager();

      //----- Define a line segment
      G4Polyline polyline;
      G4double charge = pSM->GetTrack()->GetDefinition()->GetPDGCharge();
      G4Colour colour;
      if      (charge < 0.) colour = G4Colour(1., 0., 0.);
      else if (charge < 0.) colour = G4Colour(0., 0., 1.);
      else                  colour = G4Colour(0., 1., 0.);
      G4VisAttributes attribs(colour);
      polyline.SetVisAttributes(attribs);
      polyline.push_back(pSM->GetStep()->GetPreStepPoint()->GetPosition());
      polyline.push_back(pSM->GetStep()->GetPostStepPoint()->GetPosition());

      //----- Call a drawing method for G4Polyline
      pVVisManager -> Draw(polyline);
    }
}
```

As well as tracking steps, you can visualize any kind 3D object made of line segments, using class `G4Polyline` and its drawing method, defined in class `G4VVisManager`. See, for example, the implementation of the `/vis/scene/add/axes` command.

## 8.5.10 Visualization User Actions

You can implement the `Draw` method of `G4VUserVisAction`, e.g., the class definition could be:

```
class MyVisAction: public G4VUserVisAction {
  void Draw();
};
```

and the implementation:

```
void MyVisAction::Draw() {
  G4VVisManager* pVisManager = G4VVisManager::GetConcreteInstance();
  if (pVisManager) {
```

(continues on next page)

(continued from previous page)

```
    // Simple box...
    pVisManager->Draw(G4Box("box",2*m,2*m,2*m),
                      G4VisAttributes(G4Colour(1,1,0)));

    // Etc...
  }
}
```

If efficiency is an issue, create the objects in the constructor, delete them in the destructor and draw them in your `Draw` method.

Anyway, an instance of your class needs to be registered with the vis manager, e.g.:

```
 ...
G4VisManager* visManager = new G4VisExecutive;
visManager->RegisterRunDurationUserVisAction
  ("My drawings",
   new MyVisAction,
   G4VisExtent(-10*m,10*m,-10*m,10*m,-10*m,10*m));  // This 3rd argument is optional.
visManager->Initialize ();
 ...
```

Any number of actions may be registered either as "run duration" (i.e., permanent, or at least as permanent as detector geometry) or "end of event" or "end of run" through the methods `RegisterRunDurationUserVisAction`, `RegisterEndOfEventUserVisAction` or `RegisterEndOfRunUserVisAction`. The vis manager will invoke your `Draw` method as appropriate to give your drawing a sort of permanence, for example, that can be drawn from several different angles.

Vis action drawing must be activated by adding to a scene, e.g:

```
/control/verbose 2
/vis/verbose parameters
/vis/open OGL
/vis/scene/create
/vis/scene/add/userAction
/vis/scene/add/axes
/vis/scene/add/scale
/vis/sceneHandler/attach
/vis/viewer/flush
```

The "extent" can be added on registration or on the command line or neither (if the extent of the scene is set by other components). Your `Draw` method will be called whenever needed to refresh the screen or rebuild a graphics database, for any chosen viewer. The scene can be attached to any scene handler and your drawing will be shown.

## 8.5.11 Standalone Visualization

The above raises the possibility of using GEANT4 as a "standalone" graphics package without invoking the run manager. The following main program (from `examples/extended/visualization/standalone`), together with a user vis action and a macro file–see above–will allow you to view your drawing interactively on any of the supported graphics systems:

```
#include "globals.hh"
#include "G4VisExecutive.hh"
#include "G4VisExtent.hh"
#include "G4UImanager.hh"
#include "G4UIExecutive.hh"
#include "G4SystemOfUnits.hh"
```

(continues on next page)

(continued from previous page)

```
#include "StandaloneVisAction.hh"

int main(int argc,char** argv) {

  G4UIExecutive* ui = new G4UIExecutive(argc, argv);

  G4VisManager* visManager = new G4VisExecutive;
  visManager->RegisterRunDurationUserVisAction
    ("A standalone example - 3 boxes, 2 with boolean subtracted cutout",
     new StandaloneVisAction,
     G4VisExtent(-10*m,10*m,-10*m,10*m,-10*m,10*m));
  visManager->Initialize ();

  G4UImanager::GetUIpointer()->ApplyCommand ("/control/execute standalone.mac");
  ui->SessionStart();

  delete ui;
  delete visManager;
}
```

### 8.5.12 Drawing a solid as a cloud of points

SolidCloudVisAction.hh:

```
#ifndef SOLIDCLOUDVISACTION_HH
#define SOLIDCLOUDVISACTION_HH
#include "G4VUserVisAction.hh"
#include "G4Polymarker.hh"
class G4VSolid;
class SolidCloudVisAction: public G4VUserVisAction {
public:
  SolidCloudVisAction(G4VSolid*,G4int nPoints);
  virtual void Draw();
private:
  G4Polymarker fPolymarker;
};
#endif
```

SolidCloudVisAction.cc:

```
#include "SolidCloudVisAction.hh"
#include "G4VVisManager.hh"
#include "G4VSolid.hh"
SolidCloudVisAction::SolidCloudVisAction(G4VSolid* solid, G4int nPoints).
{
  fPolymarker.SetMarkerType(G4Polymarker::dots);
  fPolymarker.SetSize(G4VMarker::screen,1.);
  for (G4int i = 0; i < nPoints; ++i) {
    G4ThreeVector p = solid->GetPointOnSurface();
    G4cout << solid->GetName() << " " << p << G4endl;
    fPolymarker.push_back(p);
  }
}
void SolidCloudVisAction::Draw() {
  G4VVisManager* pVisManager = G4VVisManager::GetConcreteInstance();
  if (pVisManager) pVisManager->Draw(fPolymarker);
}
```

Then just after you instantiate the vis manager:

```
G4VSolid* torus = new G4Torus("Torus",2.*cm,5.*cm,6.*cm,0.,CLHEP::twopi);
visManager->RegisterRunDurationUserVisAction
```

(continues on next page)

(continued from previous page)

```
("Torus",
 new SolidCloudVisAction(torus,100000),
 torus->GetExtent());
```

Then on the command line:

```
/vis/scene/create
/vis/scene/add/userAction Torus
/vis/sceneHandler/attach
```

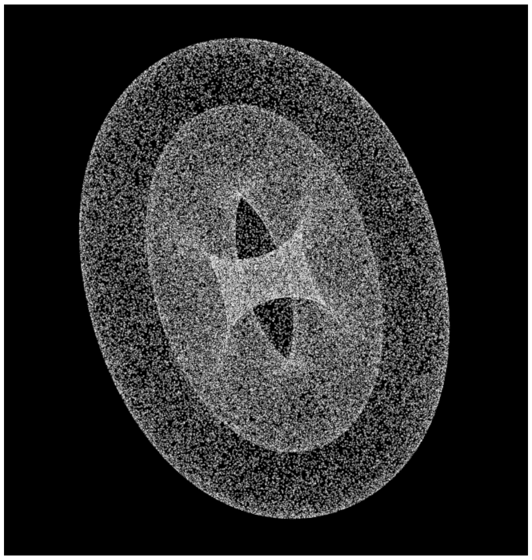


Fig. 8.3: A torus represented by a cloud of points on its surface.

## 8.6 Visualization Attributes

Visualization attributes are extra pieces of information associated with the visualizable objects. This information is necessary only for visualization, and is not included in geometrical information such as shapes, position, and orientation. Typical examples of visualization attributes are Color, Visible/Invisible, Wireframe/Solid. For example, in visualizing a box, the Visualization Manager must know its colour. If an object to be visualized has not been assigned a set of visualization attributes, then an appropriate default set is used automatically.

A set of visualization attributes is held by an instance of class `G4VisAttributes` defined in the `graphics_reps` category. In the following, we explain the main fields of the `G4VisAttributes` one by one.

### 8.6.1 Visibility

Visibility is a Boolean flag to control the visibility of objects that are passed to the Visualization Manager for visualization. Visibility is set with the following access function:

```
void G4VisAttributes::SetVisibility (G4bool visibility);
```

If you give `false` to the argument, and if culling is activated (see below), visualization is skipped for objects for which this set of visualization attributes is assigned. The default value of visibility is `true`.

Note that whether an object is visible or not is also affected by the current culling policy, which can be tuned with visualization commands.

By default the following public static function is defined:

```
static const G4VisAttributes& GetInvisible();
```

which returns a reference to a const object in which visibility is set to `false`. It can be used as follows:

```
experimentalHall_logical -> SetVisAttributes (G4VisAttributes::GetInvisible());
```

Direct access to the public static const data member `G4VisAttributes::Invisible` is also possible but deprecated on account of initialisation issues with dynamic libraries.

## 8.6.2 Colour

### Construction

Class `G4Colour` (an equivalent class name, `G4Color`, is also available) has 4 fields, which represent the RGBA (red, green, blue, and alpha) components of colour. Each component takes a value between 0 and 1. If an irrelevant value, i.e., a value less than 0 or greater than 1, is given as an argument of the constructor, such a value is automatically clipped to 0 or 1. Alpha is opacity. Its default value `1` means "opaque".

A `G4Colour` object is instantiated by giving red, green, and blue components to its constructor, i.e.,

```
G4Colour::G4Colour ( G4double r = 1.0,
                     G4double g = 1.0,
                     G4double b = 1.0,
                     G4double a = 1.0);
                             // 0<=red, green, blue <= 1.0
```

The default value of each component is 1.0. That is to say, the default colour is "white" (opaque).

For example, colours which are often used can be instantiated as follows:

```
G4Colour  white   ()              ;  // white
G4Colour  white   (1.0, 1.0, 1.0) ;  // white
G4Colour  gray    (0.5, 0.5, 0.5) ;  // gray
G4Colour  black   (0.0, 0.0, 0.0) ;  // black
G4Colour  red     (1.0, 0.0, 0.0) ;  // red
G4Colour  green   (0.0, 1.0, 0.0) ;  // green
G4Colour  blue    (0.0, 0.0, 1.0) ;  // blue
G4Colour  cyan    (0.0, 1.0, 1.0) ;  // cyan
G4Colour  magenta (1.0, 0.0, 1.0) ;  // magenta
G4Colour  yellow  (1.0, 1.0, 0.0) ;  // yellow
```

It is also possible to instantiate common colours through static public data member functions:

```
static const G4Colour& White   ();
static const G4Colour& Gray    ();
static const G4Colour& Grey    ();
static const G4Colour& Black   ();
static const G4Colour& Red     ();
static const G4Colour& Green   ();
static const G4Colour& Blue    ();
static const G4Colour& Cyan    ();
static const G4Colour& Magenta ();
static const G4Colour& Yellow  ();
```

For example, a local `G4Colour` could be constructed as:

```
G4Colour myRed(G4Colour::Red());
```

After instantiation of a `G4Colour` object, you can access to its components with the following access functions:

```
G4double G4Colour::GetRed   () const ; // Get the red   component.
G4double G4Colour::GetGreen () const ; // Get the green component.
G4double G4Colour::GetBlue  () const ; // Get the blue  component.
```

## Colour Map

`G4Colour` also provides a static colour map, giving access to predefined `G4Colour`'s through a `G4String` key. The default mapping is:

```
G4String           G4Colour
---------------------------------------
white              G4Colour::White   ()
gray               G4Colour::Gray    ()
grey               G4Colour::Grey    ()
black              G4Colour::Black   ()
red                G4Colour::Red     ()
green              G4Colour::Green   ()
blue               G4Colour::Blue    ()
cyan               G4Colour::Cyan    ()
magenta            G4Colour::Magenta ()
yellow             G4Colour::Yellow  ()
```

Colours can be retrieved through the GetColour method:

```
bool G4Colour::GetColour(const G4String& key, G4Colour& result)
```

For example:

```
G4Colour myColour(G4Colour::Black());
if (G4Colour::GetColour("red", myColour)) {
  // Successfully retrieved colour "red". myColour is now red
}
else {
  // Colour did not exist in map. myColour is still black
}
```

To see a list of available named colours, `/vis/list`. These names may also be used to specify colours in many `/vis` commands.

If the key is not registered in the colour map, a warning message is printed and the input colour is not changed. The colour map is case insensitive.

It is also possible to load user defined `G4Colour`'s into the map through the public AddToMap method. For example:

```
G4Colour myColour(0.2, 0.2, 0.2, 1);
G4Colour::AddToMap("custom", myColour);
```

This loads a user defined `G4Colour` with key "custom" into the colour map.

It is also possible to use the colours in g4tools:

```
#include "tools/colors"
...
G4Colour niceColour = tools::get_color_aquamarine<G4Colour>();
```

### Colour and G4VisAttributes

Class `G4VisAttributes` holds its colour entry as an object of class `G4Colour`. A `G4Colour` object is passed to a `G4VisAttributes` object with the following access functions:

```
//----- Set functions of G4VisAttributes.
void G4VisAttributes::SetColour (const G4Colour& colour);
void G4VisAttributes::SetColor (const G4Color& color );
```

We can also set RGBA components directly:

```
//----- Set functions of G4VisAttributes
void G4VisAttributes::SetColour ( G4double red   ,
                                  G4double green ,
                                  G4double blue  ,
                                  G4double alpha = 1.0);

void G4VisAttributes::SetColor  ( G4double red   ,
                                  G4double green ,
                                  G4double blue  ,
                                  G4double alpha = 1.);
```

The following constructor with `G4Colour` as its argument is also supported:

```
//----- Constructor of G4VisAttributes
G4VisAttributes::G4VisAttributes (const G4Colour& colour);
```

Note that colour assigned to a `G4VisAttributes` object is not always the colour that ultimately appears in the visualization. The ultimate appearance may be affected by shading and lighting models applied in the selected visualization driver or stand-alone graphics system.

## 8.6.3 Forcing attributes

As you will see later, you can select a “drawing style” from various options. For example, you can select your detector components to be visualized in “wireframe” or with “surfaces”. In the former, only the edges of your detector are drawn and so the detector looks transparent. In the latter, your detector looks opaque with shading effects.

The forced wireframe and forced solid styles make it possible to mix the wireframe and surface visualization (if your selected graphics system supports such visualization). For example, you can make only the outer wall of your detector “wired” (transparent) and can see inside in detail.

Forced wireframe style is set with the following access function:

```
void G4VisAttributes::SetForceWireframe (G4bool force);
```

If you give `true` as the argument, objects for which this set of visualization attributes is assigned are always visualized in wireframe even if in general, the surface drawing style has been requested. The default value of the forced wireframe style is `false`.

Similarly, forced solid style, i.e., to force that objects are always visualized with surfaces, is set with:

```
void G4VisAttributes::SetForceSolid (G4bool force);
```

The default value of the forced solid style is `false`, too.

You can also force auxiliary edges to be visible. Normally they are not visible unless you set the appropriate view parameter. Forcing the auxiliary edges to be visible means that auxiliary edges will be seen whatever the view parameters.

Auxiliary edges are not genuine edges of the volume. They may be in a curved surface made out of polygons, for example, or in plane surface of complicated shape that has to be broken down into simpler polygons. HepPolyhedron breaks all surfaces into triangles or quadrilaterals. There will be auxiliary edges for any volumes with a curved surface, such as a tube or a sphere, or a volume resulting from a Boolean operation. Normally, they are not shown, but sometimes it is useful to see them. In particular, a sphere, because it has no edges, will not be seen in wireframe mode in some graphics systems unless requested by the view parameters or forced, as described here.

To force auxiliary edges to be visible, use:

```
void G4VisAttributes::SetForceAuxEdgeVisible (G4bool force);
```

The default value of the force auxiliary edges visible flag is `false`.

For volumes with edges that are parts of a circle, such as a tube (G4Tubs), etc., it is possible to force the precision of polyhedral representation for visualisation. This is recommended for volumes containing only a small angle of circle, for example, a thin tube segment.

For visualisation, a circle is represented by an N-sided polygon. The default is 24 sides or segments. The user may change this for all volumes in a particular viewer at run time with /vis/viewer/set/lineSegmentsPerCircle; alternatively it can be forced for a particular volume with:

```
void G4VisAttributes::SetForceLineSegmentsPerCircle (G4int nSegments);
```

## 8.6.4 Other attributes

Here is a list of Set methods for class `G4VisAttributes`:

```
void SetVisibility           (G4bool);
void SetDaughtersInvisible   (G4bool);
void SetColour               (const G4Colour&);
void SetColor                (const G4Color&);
void SetColour               (G4double red, G4double green, G4double blue,
                              G4double alpha = 1.);
void SetColor                (G4double red, G4double green, G4double blue,
                              G4double alpha = 1.);
void SetLineStyle            (LineStyle);
void SetLineWidth            (G4double);
void SetForceWireframe       (G4bool);
void SetForceSolid           (G4bool);
void SetForceAuxEdgeVisible  (G4bool);
void SetForceLineSegmentsPerCircle (G4int nSegments);
// Allows choice of circle approximation.  A circle of 360 degrees
// will be composed of nSegments line segments.  If your solid has
// curves of D degrees that you need to divide into N segments,
// specify nSegments = N * 360 / D.
void SetStartTime            (G4double);
void SetEndTime              (G4double);
void SetAttValues            (const std::vector<G4AttValue>*);
void SetAttDefs              (const std::map<G4String,G4AttDef>*);
```

### 8.6.5 Constructors of G4VisAttributes

The following constructors are supported for class `G4VisAttributes`:

```
//----- Constructors of class G4VisAttributes
G4VisAttributes (void);
G4VisAttributes (G4bool visibility);
G4VisAttributes (const G4Colour& colour);
G4VisAttributes (G4bool visibility, const G4Colour& colour);
```

### 8.6.6 How to assign G4VisAttributes to a logical volume

In constructing your detector components, you may assign a set of visualization attributes to each "logical volume" in order to visualize them later (if you do not do this, the graphics system will use a default set). You cannot make a solid such as `G4Box` hold a set of visualization attributes; this is because a solid should hold only geometrical information. At present, you cannot make a physical volume hold one, but there are plans to design a memory-efficient way to do it; however, you can visualize a transient piece of solid or physical volume with a temporary assigned set of visualization attributes.

Class `G4LogicalVolume` holds a pointer of `G4VisAttributes.` This field is set and referenced with the following access functions:

```
//----- Set functions of G4VisAttributes
void G4VisAttributes::SetVisAttributes (const G4VisAttributes* pVA);
void G4VisAttributes::SetVisAttributes (const G4VisAttributes& VA);

//----- Get functions of G4VisAttributes
const G4VisAttributes* G4VisAttributes::GetVisAttributes () const;
```

The following is sample C++ source codes for assigning a set of visualization attributes with cyan colour and forced wireframe style to a logical volume:

```
//----- C++ source codes: Assigning G4VisAttributes to a logical volume
...
     // Instantiation of a logical volume
myTargetLog = new G4LogicalVolume( myTargetTube,BGO, "TLog", 0, 0, 0);
...
     // Instantiation of a set of visualization attributes with cyan colour
G4VisAttributes * calTubeVisAtt = new G4VisAttributes(G4Colour(0.,1.,1.));
     // Set the forced wireframe style
calTubeVisAtt->SetForceWireframe(true);
     // Assignment of the visualization attributes to the logical volume
myTargetLog->SetVisAttributes(calTubeVisAtt);
```

Note that the life of the visualization attributes must be at least as long as the objects to which they are assigned; it is the users' responsibility to ensure this, and to delete the visualization attributes when they are no longer needed (or just leave them to die at the end of the job).

### 8.6.7 Additional User-Defined Attributes

GEANT4 Trajectories and Hits can be assigned additional arbitrary attributes:

- they can be displayed when you click on the relevant object in the FRED or HepRApp browsers. For example, HepRApp then lets you pick objects and see their attributes or select visibility based on these attributes.
- they can be used for filtering of trajectories, hits and digis (see *Trajectory Filtering*).

Define the attributes with lines such as:

```
std::map<G4String,G4AttDef>* store = G4AttDefStore::GetInstance("G4Trajectory",isNew);
G4String PN("PN");
(*store)[PN] = G4AttDef(PN,"Particle Name","Physics","","G4String");
G4String IMom("IMom");
(*store)[IMom] = G4AttDef(IMom, "Momentum of track at start of trajectory", "Physics", "",
                                "G4ThreeVector");
```

Then fill the attributes with lines such as:

```
std::vector<G4AttValue>* values = new std::vector<G4AttValue>;
values->push_back(G4AttValue("PN",ParticleName,""));
s.seekp(std::ios::beg);
s << G4BestUnit(initialMomentum,"Energy") << std::ends;
values->push_back(G4AttValue("IMom",c,""));
```

See geant4/source/tracking/src/G4Trajectory.cc for a good example.

`G4AttValue` objects are light, containing just the value; for the long description and other sharable information the `G4AttValue` object refers to a `G4AttDef` object. They are based on the HepRep standard described at http://www.slac.stanford.edu/~perl/heprep/. GEANT4 also provides a way of checking the results; at the point where all the data members have been set (perhaps in your sensitive detector), for example for `hit`:

```
G4cout << G4AttCheck(hit->CreateAttValues(),hit->GetAttDefs()) << G4endl;
```

GEANT4 provides some default examples of the use of this facility in the trajectory classes in /source/tracking such as `G4Trajectory`, `G4SmoothTrajectory`. `G4Trajectory::CreateAttValues` shows how `G4AttValue` objects can be made and `G4Trajectory::GetAttDefs` shows how to make the corresponding `G4AttDef` objects and use the `G4AttDefStore`. Note that the "user" of CreateAttValues guarantees to destroy them; this is a way of allowing creation on demand and leaving the `G4Trajectory` object, for example, free of such objects in memory. The comments in `G4VTrajectory.hh` explain further and additional insights might be obtained by looking at two methods which use them, namely `G4VTrajectory::DrawTrajectory` and `G4VTrajectory::ShowTrajectory`.

Hits classes in examples /extended/analysis/A01 and /extended/runAndEvent/RE01 show how to do the same for your hits. The base class no-action methods CreateAttValues and GetAttDefs should be overridden in your concrete class. The comments in `G4VHit.hh` explain further.

In addition, the user is free to add a `G4std::vector<G4AttValue>*` and a `G4std::vector<G4AttDef>*` to a `G4VisAttributes` object as could, for example, be used by a `G4LogicalVolume` object.

At the time of writing, only the HepRep graphics systems are capable of displaying the G4AttValue information, but this information will become useful for all GEANT4 visualization systems through improvements in release 8.1 or later.

## 8.7 Enhanced Trajectory Drawing

### 8.7.1 Default Configuration

Trajectory drawing styles are specified through trajectory drawing models. Each drawing model has a default configuration provided through a G4VisTrajContext object. The default context settings are shown below.

Table 8.4: Default context settings for trajectory drawing models.

| Property | Default Setting |
|---|---|
| Line colour | grey |
| Line visibility | true |
| Draw line | true |
| Draw auxiliary points | false |
| Auxiliary point type | squares |
| Auxiliary point size | 2 pixels or mm* |
| Auxiliary point size type | screen |
| Auxiliary point fill style | filled |
| Auxiliary point colour | magenta |
| Auxiliary point visibility | true |
| Draw step point | false |
| Step point type | circles |
| Step point size | 2 pixels or mm* |
| Step point size type | screen |
| Step point fill style | filled |
| Step point colour | yellow |
| Step point visibility | true |
| Time slice interval | 0 |

* Depending on size type. If size type == screen, pixels are assumed and no unit need be supplied. If size type == world, a unit must be supplied, e.g., 10 cm.

**Note:** Different visualisation drivers handle trajectory configuration in different ways, so trajectories may not necessarily get displayed as you have configured them.

### 8.7.2 Trajectory Drawing Models

A trajectory drawing model can override the default context according to the properties of a given trajectory. The following models are supplied with the GEANT4 distribution:

- G4TrajectoryGenericDrawer (generic)
- G4TrajectoryDrawByCharge (drawByCharge)
- G4TrajectoryDrawByParticleID (drawByParticleID)
- G4TrajectoryDrawByOriginVolume (drawByOriginVolume)
- G4TrajectoryDrawByTouchedVolume (drawByTouchedVolume)
- G4TrajectoryDrawByAttribute (drawByAttribute)

Both the context and model properties can be configured by the user. The models are described briefly below, followed by some example configuration commands.

### G4TrajectoryGenericDrawer

This model simply draws all trajectories in the same style, with the properties provided by the context.

### G4TrajectoryDrawByCharge

This is the default model - if no model is specified by the user, this model will be constructed automatically. The trajectory lines are coloured according to charge, with all other configuration parameters provided by the default context. The default colouring scheme is shown below.

| Charge | Colour |
|---|---|
| 1 | blue |
| -1 | red |
| 0 | green |

### G4TrajectoryDrawByParticleID

This model colours trajectory lines according to particle type. All other configuration parameters are provided by the default context. Chosen particle types can be highlighted with specified colours. By default, trajectories are coloured according to the scheme below and any other particle in the default colour (grey). (All may be overridden by the set command.)

| Particle | Colour |
|---|---|
| gamma | green |
| e- | red |
| e+ | blue |
| pi+ | magenta |
| pi- | magenta |
| proton | cyan |
| neutron | yellow |

### G4TrajectoryDrawByOriginVolume

This model colours trajectory lines according to the trajectory's originating volume name. The volume can be either a logical or physical volume. Physical volume takes precedence over logical volume. All trajectories are coloured grey by default.

### G4TrajectoryDrawByTouchedVolume

This model colours trajectory lines if it touches one or more volumes according to the physical volume name(s). It requires rich trajectories, G4RichTrajectory (`/vis/scene/add/trajectories rich`). All trajectories are coloured grey by default.

### G4TrajectoryDrawByAttribute

This model draws trajectories based on the HepRep style attributes associated with trajectories. Each attribute drawer can be configured with interval and/or single value data. A new context object is created for each interval/single value. This makes it possible to have different step point markers etc, as well as line colour for trajectory attributes falling into different intervals, or matching single values. The single value data should override the interval data, allowing specific values to be highlighted. Units should be specified on the command line if the attribute unit is specified either as a G4BestUnit or if the unit is part of the value string.

## 8.7.3 Controlling from Commands

Multiple trajectory models can be created and configured using commands in the "`/vis/modeling/trajectories/`" directory. It is then possible to list available models and select one to be current.

Model configuration commands are generated dynamically when a model is instantiated. These commands apply directly to that instance. This makes it possible to have multiple instances of the drawByCharge model for example, each independently configurable through it's own set of commands.

See the interactive help for more information on the available commands.

### Example commands

- Create a generic model named generic-0 by default

```
/vis/modeling/trajectories/create/generic
```

- Configure context to colour all trajectories red

```
/vis/modeling/trajectories/generic-0/default/setLineColour red
```

- Create a drawByCharge model named drawCharge-0 by default (Subsequent models will be named drawByCharge-1, drawByCharge-2, etc.)

```
/vis/modeling/trajectories/create/drawByCharge
```

- Create a drawByCharge model named testChargeModel

```
/vis/modeling/trajectories/create/drawByCharge testChargeModel
```

- Configure drawByCharge-0 model

```
/vis/modeling/trajectories/drawByCharge-0/set 1 red
/vis/modeling/trajectories/drawByCharge-0/set -1 red
/vis/modeling/trajectories/drawByCharge-0/set 0 white
```

- Configure testCharge model through G4Colour components

```
/vis/modeling/trajectories/testChargeModel/setRGBA 1 0 1 1 1
/vis/modeling/trajectories/testChargeModel/setRGBA -1 0.5 0.5 0.5 1
/vis/modeling/trajectories/testChargeModel/setRGBA 0 1 1 0 1
```

- Create a drawByParticleID model named drawByParticleID-0

```
/vis/modeling/trajectories/create/drawByParticleID
```

- Configure drawByParticleID-0 model

```
/vis/modeling/trajectories/drawByParticleID-0/set gamma red
/vis/modeling/trajectories/drawByParticleID-0/setRGBA e+ 1 0 1 1
```

- List available models

```
/vis/modeling/trajectories/list
```

- select drawByParticleID-0 to be current

```
/vis/modeling/trajectories/select drawByParticleID-0
```

- Create a drawByAttribute model named drawByAttribute-0

```
/vis/modeling/trajectories/create/drawByAttribute
```

- Configure drawByAttribute-0 model

```
/vis/modeling/trajectories/drawByAttribute-0/verbose true
```

- Select attribute "CPN"

```
/vis/modeling/trajectories/drawByAttribute-0/setAttribute CPN
```

- Configure single value data

```
/vis/modeling/trajectories/drawByAttribute-0/addValue brem_key  eBrem
/vis/modeling/trajectories/drawByAttribute-0/addValue annihil_key annihil
/vis/modeling/trajectories/drawByAttribute-0/addValue decay_key Decay
/vis/modeling/trajectories/drawByAttribute-0/addValue muIon_key muIoni
/vis/modeling/trajectories/drawByAttribute-0/addValue eIon_key  eIoni

/vis/modeling/trajectories/drawByAttribute-0/brem_key/setLineColour     red
/vis/modeling/trajectories/drawByAttribute-0/annihil_key/setLineColour  green
/vis/modeling/trajectories/drawByAttribute-0/decay_key/setLineColour    cyan
/vis/modeling/trajectories/drawByAttribute-0/eIon_key/setLineColour     yellow
/vis/modeling/trajectories/drawByAttribute-0/muIon_key/setLineColour magenta
```

- Create a drawByAttribute model named drawByAttribute-1

```
/vis/modeling/trajectories/create/drawByAttribute
```

- Select "IMag" attribute

```
/vis/modeling/trajectories/drawByAttribute-1/setAttribute IMag
```

- Configure interval data

```
/vis/modeling/trajectories/drawByAttribute-1/addInterval interval1 0.0 keV 2.5MeV
/vis/modeling/trajectories/drawByAttribute-1/addInterval interval2 2.5 MeV 5 MeV
/vis/modeling/trajectories/drawByAttribute-1/addInterval interval3 5 MeV 7.5 MeV
/vis/modeling/trajectories/drawByAttribute-1/addInterval interval4 7.5 MeV 10 MeV
/vis/modeling/trajectories/drawByAttribute-1/addInterval interval5 10 MeV 12.5 MeV
/vis/modeling/trajectories/drawByAttribute-1/addInterval interval6 12.5 MeV 10000 MeV

/vis/modeling/trajectories/drawByAttribute-1/interval1/setLineColourRGBA 0.8 0 0.8 1
/vis/modeling/trajectories/drawByAttribute-1/interval2/setLineColourRGBA 0.23 0.41 1 1
/vis/modeling/trajectories/drawByAttribute-1/interval3/setLineColourRGBA 0 1 0 1
/vis/modeling/trajectories/drawByAttribute-1/interval4/setLineColourRGBA 1 1 0 1
/vis/modeling/trajectories/drawByAttribute-1/interval5/setLineColourRGBA 1 0.3 0 1
/vis/modeling/trajectories/drawByAttribute-1/interval6/setLineColourRGBA 1 0 0 1
```

- Create a drawByEncounteredVolume model named drawByEncounteredVolume-0

```
/vis/modeling/trajectories/create/drawByEncounteredVolume
```

- Change the color for a specific encountered shape

```
/vis/modeling/trajectories/drawByEncounteredVolume-0/set Shape1 cyan
```

### 8.7.4 Controlling from Compiled Code

It is possible to use the enhanced trajectory drawing functionality in compiled code as well as from commands. Multiple trajectory models can be instantiated, configured and registered with G4VisManager. Once registered, the models are owned by G4VisManager, and must not be deleted by the user.

Only one model may be current. For example:

```
G4VisManager* visManager = new G4VisExecutive;
visManager->Initialize();

G4TrajectoryDrawByParticleID* model = new G4TrajectoryDrawByParticleID;
G4TrajectoryDrawByParticleID* model2 = new G4TrajectoryDrawByParticleID("test");

model->SetDefault("cyan");
model->Set("gamma", "green");
model->Set("e+", "magenta");
model->Set("e-", G4Colour(0.3, 0.3, 0.3));

visManager->RegisterModel(model);
visManager->RegisterModel(model2);

visManager->SelectTrajectoryModel(model->Name());
```

### 8.7.5 Drawing by time

To draw by time, you need to use G4RichTrajectory, for example:

```
/vis/scene/add/trajectories rich
```

or

```
/vis/scene/add/trajectories rich smooth
```

When you run, you need to create a trajectory model and set the time slice interval (remembering that particles are often relativistic and travel 30 cm/ns):

```
/vis/modeling/trajectories/create/drawByCharge
/vis/modeling/trajectories/drawByCharge-0/default/setDrawStepPts true
/vis/modeling/trajectories/drawByCharge-0/default/setStepPtsSize 5
/vis/modeling/trajectories/drawByCharge-0/default/setDrawAuxPts true
/vis/modeling/trajectories/drawByCharge-0/default/setAuxPtsSize 5
/vis/modeling/trajectories/drawByCharge-0/default/setTimeSliceInterval 0.1 ns
/vis/modeling/trajectories/list
```

and use a graphics driver that can display by time:

```
/vis/open OGL
/vis/drawVolume
/vis/scene/add/trajectories rich
/vis/viewer/set/timeWindow/startTime 0.5 ns
/vis/viewer/set/timeWindow/endTime 0.8 ns
/run/beamOn
/vis/viewer/refresh
```

For tips on how to see particles “moving through time” see *Making a Movie*.

## 8.8 Trajectory Filtering

Trajectory filtering allows you to visualise a subset of available trajectories. This can be useful if you only want to view interesting trajectories and discard uninteresting ones. Trajectory filtering can be run in two modes:

- **Soft filtering**: In this mode, uninteresting trajectories are marked invisible. Hence, they are still written, but (depending on the driver) will not be displayed. Some drivers, for example the HepRepFile driver, which writes an XML file for the HepRApp browser, will allow you to selectively view these soft filtered trajectories.
- **Hard filtering** (default): In this mode, uninteresting trajectories are not drawn at all. This mode is especially useful if the job produces huge graphics files, dominated by data from uninteresting trajectories. It is essential to use this mode for most drivers, including those using OpenGL, because they are not adapted for soft filtering.

Change mode with `/vis/filtering/trajectories/mode`.

Trajectory filter models are used to apply filtering according to specific criteria. The following models are currently supplied with the GEANT4 distribution:

- G4TrajectoryChargeFilter (chargeFilter)
- G4TrajectoryParticleFilter (particleFilter)
- G4TrajectoryOriginVolumeFilter (originVolumeFilter)
- G4TrajectoryTouchedVolumeFilter (touchedVolumeFilter)
- G4TrajectoryAttributeFilter (attributeFilter)

Multiple filters are automatically chained together, and can configured either interactively or in commands or in compiled code. The filters can be inverted, set to be inactive or set in a verbose mode. The above models are described briefly below, followed by some example configuration commands.

**G4TrajectoryChargeFilter**

This model filters trajectories according to charge. In standard running mode, only trajectories with charges matching those registered with the model will pass the filter.

**G4TrajectoryParticleFilter**

This model filters trajectories according to particle type. In standard running mode, only trajectories with particle types matching those registered with the model will pass the filter.

**G4TrajectoryOriginVolumeFilter**

This model filters trajectories according to originating volume name. In standard running mode, only trajectories with originating volumes matching those registered with the model will pass the filter.

**G4TrajectoryTouchedVolumeFilter**

This model filters trajectories that touch one or more volumes according to the physical volume name(s). It requires rich trajectories, G4RichTrajectory (`/vis/scene/add/trajectories rich`). In standard running mode, only trajectories that touch volumes matching those registered with the model will pass the filter.

**G4TrajectoryAttributeFilter**

This model filters trajectories based on the HepRep style attributes (see *Additional User-Defined Attributes*) associated with trajectories. Each attribute drawer can be configured with interval and/or single value data. Single value data should override the interval data. Units should be specified on the command line if the attribute unit is specified either as a G4BestUnit or if the unit is part of the value string. Available attributes can be shown with `/vis/list`.

### 8.8.1 Controlling from Commands

Multiple trajectory filter models can be created and configured using commands in the "`/vis/filtering/trajectories/`" directory. All generated filter models are chained together automatically.

Model configuration commands are generated dynamically when a filter model is instantiated. These commands apply directly to that instance.

See the interactive help for more information on the available commands.

### 8.8.2 Example commands

```
# Create a particle filter. Configure to pass only gammas. Then
# invert to pass anything other than gammas. Set verbose printout,
# and then deactivate filter

/vis/filtering/trajectories/create/particleFilter
/vis/filtering/trajectories/particleFilter-0/add gamma
/vis/filtering/trajectories/particleFilter-0/invert true
/vis/filtering/trajectories/particleFilter-0/verbose true
/vis/filtering/trajectories/particleFilter-0/active false
```

```
# Create a charge filter. Configure to pass only neutral trajectories.
# Set verbose printout. Reset filter and reconfigure to pass only
# negatively charged trajectories.

/vis/filtering/trajectories/create/chargeFilter
/vis/filtering/trajectories/chargeFilter-0/add 0
/vis/filtering/trajectories/chargeFilter-0/verbose true
/vis/filtering/trajectories/chargeFilter-0/reset true
/vis/filtering/trajectories/chargeFilter-0/add -1
```

```
# Create an attribute filter named attributeFilter-0
/vis/filtering/trajectories/create/attributeFilter

# Select attribute "IMag"
/vis/filtering/trajectories/attributeFilter-0/setAttribute IMag

# Select trajectories with 2.5 MeV <= IMag< 1000 MeV
/vis/filtering/trajectories/attributeFilter-0/addInterval 2.5 MeV 1000 MeV
```

```
# List available attributes
/vis/list

# List filters
/vis/filtering/trajectories/list
```

Note that although `particleFilter-0` and `chargeFilter-0` are automatically chained, `particleFilter-0` will not have any effect since it is has been deactivated.

### 8.8.3 Hit and Digi Filtering

The attribute based filtering can be used on hits and digitisations as well as trajectories. To active the interactive attribute based hit filtering, a filter call should be added to the "Draw" method of the hit (or digi) class:

```
void MyHit::Draw()
{
   ...
   if (! pVVisManager->FilterHit(*this)) return;
   ...
}
```

Interactive filtering can then be done through the commands in `/vis/filtering/hits` or `digi`.

## 8.9 Polylines, Markers and Text

Polylines, markers and text are defined in the `graphics_reps` category, and are used only for visualization (*Controlling Visualization from Compiled Code*). Users may create any of these objects with local scope; once drawn, they may safely be deleted or allowed to go out of scope.

### 8.9.1 Polylines

A polyline is a set of successive line segments. It is defined with a class `G4Polyline` defined in the `graphics_reps` category. A polyline is used to visualize tracking steps, particle trajectories, coordinate axes, and any other user-defined objects made of line segments.

`G4Polyline` is defined as a list of `G4Point3D` objects, i.e., vertex positions. The vertex positions are set to a `G4Polyline` object with the `push_back()` method.

For example, an x-axis with length 5 cm and with red color is defined in shown in the Listing 8.3.

Listing 8.3: Defining an x-axis with length 5 cm and with colour red.

```
//-----  An example of defining a line segment
// Instantiate an empty polyline object
G4Polyline  x_axis;

// Set red line colour
G4Colour         red(1.0, 0.0, 0.0);
G4VisAttributes  att(red);
x_axis.SetVisAttributes(&att);

// Set vertex positions
x_axis.push_back( G4Point3D(0., 0., 0.) );
x_axis.push_back( G4Point3D(5.*cm, 0., 0.) );
```

### 8.9.2 Markers

Here we explain how to use 3D markers in GEANT4 Visualization.

**What are Markers?**

Markers set marks at arbitrary positions in the 3D space. They are often used to visualize hits of particles at detector components. A marker is a 2-dimensional primitive with shape (square, circle, etc), color, and special properties (a) of always facing the camera and (b) of having the possibility of a size defined in screen units (pixels). Here "size" means "overall size", e.g., diameter of circle and side of square (but diameter and radius access functions are defined to avoid ambiguity).

So the user who constructs a marker should decide whether or not it should be visualized to a given size in world coordinates by setting the world size. Alternatively, the user can set the screen size and the marker is visualized to its screen size. Finally, the user may decide not to set any size; in that case, it is drawn according to the sizes specified in the default marker specified in the class `G4ViewParameters`.

By default, "square" and "circle" are supported in GEANT4 Visualization. The former is described with class `G4Square`, and the latter with class `G4Circle`:

| Marker Type | Class Name |
|---|---|
| circle | `G4Circle` |
| right square | `G4Square` |

These classes are inherited from class `G4VMarker`. They have constructors as follows:

```
//----- Constructors of G4Circle and G4Square
G4Circle::G4Circle (const G4Point3D& pos );
G4Square::G4Square (const G4Point3D& pos);
```

Access functions of class `G4VMarker` are summarized below.

**Access functions of markers**

Listing 8.4 shows the access functions inherited from the base class `G4VMarker`.

Listing 8.4: The access functions inherited from the base class `G4VMarker`.

```
//----- Set functions of G4VMarker
void G4VMarker::SetPosition( const G4Point3D& );
void G4VMarker::SetWorldSize( G4double );
void G4VMarker::SetWorldDiameter( G4double );
```

(continues on next page)

(continued from previous page)

```
void G4VMarker::SetWorldRadius( G4double );
void G4VMarker::SetScreenSize( G4double );
void G4VMarker::SetScreenDiameter( G4double );
void G4VMarker::SetScreenRadius( G4double );
void G4VMarker::SetFillStyle( FillStyle );
// Note: enum G4VMarker::FillStyle {noFill, hashed, filled};

//----- Get functions of G4VMarker
G4Point3D G4VMarker::GetPosition () const;
G4double G4VMarker::GetWorldSize () const;
G4double G4VMarker::GetWorldDiameter () const;
G4double G4VMarker::GetWorldRadius () const;
G4double G4VMarker::GetScreenSize () const;
G4double G4VMarker::GetScreenDiameter () const;
G4double G4VMarker::GetScreenRadius () const;
FillStyle G4VMarker::GetFillStyle () const;
// Note: enum G4VMarker::FillStyle {noFill, hashed, filled};
```

Listing 8.5 shows sample C++ source code to define a very small red circle, i.e., a dot with diameter 1.0 pixel. Such a dot is often used to visualize a hit.

Listing 8.5: Sample C++ source code to define a very small red circle.

```
//----- An example of defining a red small maker
G4Circle circle(position); // Instantiate a circle with its 3D
                           // position. The argument "position"
                           // is defined as G4Point3D instance
circle.SetScreenDiameter (1.0); // Should be circle.SetScreenDiameter
                                //  (1.0 * pixels) - to be implemented
circle.SetFillStyle (G4Circle::filled); // Make it a filled circle
G4Colour colour(1.,0.,0.);              // Define red color
G4VisAttributes attribs(colour);        // Define a red visualization attribute
circle.SetVisAttributes(attribs);       // Assign the red attribute to the circle
```

### 8.9.3 Text

Text, i.e., a character string, is used to visualize various kinds of description, particle name, energy, coordinate names etc. Text is described by the class `G4Text` . The following constructors are supported:

```
//----- Constructors of G4Text
G4Text (const G4String& text);
G4Text (const G4String& text, const G4Point3D& pos);
```

where the argument `text` is the text (string) to be visualized, and `pos` is the 3D position at which the text is visualized.

Text is currently drawn only by the OpenGL drivers, such as OGLIX, OGLIXm and OpenInventor. It is not yet supported on other drivers, including the Windows OpenGL drivers, HepRep, etc.

Note that class `G4Text` also inherits `G4VMarker`. Size of text is recognized as "font size", i.e., height of the text. All the access functions defined for class `G4VMarker` mentioned above are available. In addition, the following access functions are available, too:

```
//----- Set functions of G4Text
void G4Text::SetText ( const G4String& text ) ;
void G4Text::SetOffset ( double dx, double dy ) ;

//----- Get functions of G4Text
G4String G4Text::GetText () const;
G4double G4Text::GetXOffset () const;
G4double G4Text::GetYOffset () const;
```

Method `SetText()` defines text to be visualized, and `GetText()` returns the defined text. Method `SetOffset()` defines x (horizontal) and y (vertical) offsets in the screen coordinates. By default, both offsets are zero, and the text starts from the 3D position given to the constructor or to the method `G4VMarker:SetPosition()`. Offsets should be given with the same units as the one adopted for the size, i.e., world-size or screen-size units.

Listing 8.6 shows sample C++ source code to define text with the following properties:

- Text: "Welcome to Geant4 Visualization"
- Position: (0.,0.,0.) in the world coordinates
- Horizontal offset: 10 pixels
- Vertical offset: -20 pixels
- Colour: blue (default)

Listing 8.6: An example of defining text.

```
//-----  An example of defining a visualizable text

//----- Instantiation
G4Text text ;
text.SetText ( "Welcome to Geant4 Visualization");
text.SetPosition ( G4Point3D(0.,0.,0.) );
// These three lines are equivalent to:
//  G4Text text ( "Welcome to Geant4 Visualization",
//                G4Point3D(0.,0.,0.) );

//----- Size (font size in units of pixels)
G4double fontsize = 24.; // Should be 24. * pixels - to be implemented.
text.SetScreenSize ( fontsize );

//----- Offsets
G4double x_offset = 10.; // Should be 10. * pixels - to be implemented.
G4double y_offset = -20.; // Should be -20. * pixels - to be implemented.
text.SetOffset( x_offset, y_offset );

//----- Color (Blue is the default setting, and so the codes below are omittable)
G4Colour blue( 0., 0., 1. );
G4VisAttributes att ( blue );
text.SetVisAttributes ( att );
```

## 8.10 Making a Movie

These ideas are illustrated in `examples/extended/visualization/movies`.

These instructions are suggestive only. The following procedures have not been tested on all platforms. There are clearly some instructions that apply only to Unix-like systems with an X-Windows based windowing system. However, it should not be difficult to take the ideas presented here and extend them to other platforms and systems.

The procedures described here need graphics drivers that can produce picture files that can be converted to a form suitable for an MPEG encoder. There may be other ways of capturing the screen images and we would be happy to hear about them. Graphics drivers currently capable of producing picture files are:

| Driver | File type |
|---|---|
| DAWNFILE | prim then eps using **dawn** |
| HepRepFile | HepRep1 |
| OGLX | jpeg, eps, pdf, ppm, . . . |
| Qt | jpeg, eps, pdf, ppm, . . . |
| RayTracer | jpeg |
| VRML2FILE | vrml |

So far, only DAWNFILE, OGLX, OGLQt and RayTracer have been "road tested".

Once you have a set of files in a standard format, such as pdf or eps, they can be made into a movie.

- iMovie (Apple Mac only):

- Import the created files (PDF recommended).
- Reduce the cliplength to 0.1 s (that seems to be the minimum).
- Export to file.
- Play at x2.

- Convert the files to ppm with **convert** from ImageMagick, then use **ppmtompeg** from SourceForge.
- Use mpeg2encode - see *Processing picture files with mpeg2encode*.

### 8.10.1 Using `/vis/viewer/interpolate`

- Save a sequence of views with `/vis/viewer/save` (see *How to save a view.*).
- "Fly through" them with `/vis/viewer/interpolate` (OpenGL and Qt only).
- When you are happy, add the **export** parameter `/vis/viewer/interpolate ! ! ! ! export`.

The procedure is: choose a view, save, choose another view, save, and so on until you have, say, 10 saved views. Then `/vis/viewer/interpolate ! ! ! ! export` will produce hundreds of picture files.

From GEANT4 10.5, you can interpolate time windows, and make particles appear to move:

```
# See guidance on /vis/viewer/set/timeWindow/ commands.
/vis/scene/add/trajectories rich
/vis/modeling/trajectories/drawByCharge-0/default/setTimeSliceInterval 0.01 ns
/run/beamOn # or several until you get a good event or events
# Then typically
/vis/viewer/set/timeWindow/displayLightFront true 0 0 -50 cm -0.5 ns
/vis/viewer/set/timeWindow/displayHeadTime true
/vis/viewer/set/timeWindow/fadeFactor 1
/vis/viewer/set/timeWindow/startTime 0 ns 1 ns
/vis/viewer/save
/vis/viewer/set/timeWindow/startTime 1 ns 1 ns
# Then zoom, pan etc to a view of interest and
/vis/viewer/save
# Then repeat with next start time, another view and a save, then try
/vis/viewer/interpolate
# Finally
/vis/viewer/interpolate ! ! ! ! export
```

See `examples/extended/visualization/movies`.

### 8.10.2 With a macro loop

Make a macro `movie.mac` with something like this:

```
/control/verbose 2
#/run/initialize
/vis/open OGL 600x600-0+0
/vis/drawVolume
/vis/viewer/set/style surface
/vis/viewer/set/projection perspective 50 deg
/control/alias phi 30
/control/loop movie.loop theta 0 360 1
```

which invokes `movie.loop`, which is something like:

```
/vis/viewer/set/viewpointThetaPhi {theta} {phi}
/vis/ogl/export
```

Then on the command line:

```
/control/execute movie.mac
```

It may work better in "batch mode". Assuming your main program is similar to that in example B1, from your terminal:

```
./your-app-name movie.mac
```

### 8.10.3 Processing picture files with mpeg2encode

Say you have produced lots of eps files. Then. . .

```
make_mpeg2encode_parfile.sh G4OpenGL_*eps
```

Then edit mpeg2encode.par to specify file type and size, etc.:

```
$ diff mpeg2encode.par~ mpeg2encode.par
7c7
< 1         /* input picture file format: 0=*.Y,*.U,*.V, 1=*.yuv, 2=*.ppm */
---
> 2         /* input picture file format: 0=*.Y,*.U,*.V, 1=*.yuv, 2=*.ppm */
15,17c15,17
<    /* horizontal_size */
<   /* vertical_size */
< 8         /* aspect_ratio_information 1=square pel, 2=4:3, 3=16:9, 4=2.11:1 */
---
> 600   /* horizontal_size */
> 600  /* vertical_size */
> 1         /* aspect_ratio_information 1=square pel, 2=4:3, 3=16:9, 4=2.11:1 */
```

Then convert to ppm:

```
for i in G4OpenGL*eps;
  do j=`basename $i .eps`; command="convert $i $j.ppm"; echo $command; $command; done
```

Then

```
mpeg2encode mpeg2encode.par G4OpenGL.mpg
```

Then, on Mac, for example:

```
open G4OpenGL.mpg
```

opens a movie player.

### 8.10.4 Qt

The Qt driver provides one of the easiest ways to make a movie. Of course, you first need to add the Qt libraries and link with Qt, but once you have that, Qt provides a ready-made function to store all updates of the OpenGL frame into the movie format. You then use loops (as defined in OGLX section above) or even move/rotate/zoom you scene by mouse actions to form your movie.

The Qt driver automatically handles all of the movie-making steps described in the OGLX section of this document - storing the files, converting them and assembling the finished movie. You just have to take care of installing an mpeg_encoder.

To make a movie :

- Right click to display a context menu, "Action"-<"Movie parameters".
- Select MPEG encoder path if it was not found.
- Select the name of the output movie.
- Let go! Hit SPACE to Start/Pause recording, RETURN to STOP

Then, open your movie (on Mac, for example):

```
open G4OpenGL.mpg
```

opens a QuickTime player.

### 8.10.5 DAWNFILE

You need to invoke **dawn** in "direct" mode, which picks up parameters from .DAWN_1.history, and suppress the GUI:

```
alias dawn='dawn -d'
export DAWN_BATCH=1
```

Change OGL to DAWNFILE in the above set of GEANT4 commands and run. Then convert to ppm files as above:

```
for i in g4_*.eps;
  do j=`basename $i .eps`; command="convert $i $j.ppm"; echo $command; $command; done
```

Then make a .par file:

```
make_mpeg2encode_parfile.sh g4_*ppm
```

and edit mpeg2encode.par:

```
$ diff mpeg2encode.par~ mpeg2encode.par
7c7
< 1         /* input picture file format: 0=*.Y,*.U,*.V, 1=*.yuv, 2=*.ppm */
---
> 2         /* input picture file format: 0=*.Y,*.U,*.V, 1=*.yuv, 2=*.ppm */
9c9
< 1         /* number of first frame */
---
> 0         /* number of first frame */
15,16c15,16
<    /* horizontal_size */
<   /* vertical_size */
---
> 482   /* horizontal_size */
> 730  /* vertical_size */
```

Then encode and play:

```
mpeg2encode mpeg2encode.par DAWN.mpg
open DAWN.mpg
```

### 8.10.6 RayTracerX

```
/control/verbose 2
/vis/open RayTracerX 600x600-0+0
# (Raytracer doesn't need a scene; smoother not to /vis/drawVolume.)
/vis/viewer/reset
/vis/viewer/set/style surface
/vis/viewer/set/projection perspective 50 deg
/control/alias phi 30
/control/loop movie.loop theta 0 360 1
```

where movie.loop is as above. This produces lots of jpeg files (but takes 3 days!!!). Then...

```
make_mpeg2encode_parfile.sh g4RayTracer*jpeg
```

Then edit mpeg2encode.par to specify file type and size, etc.:

```
$ diff mpeg2encode.par.orig mpeg2encode.par
7c7
< 1         /* input picture file format: 0=*.Y,*.U,*.V, 1=*.yuv, 2=*.ppm */
---
> 2         /* input picture file format: 0=*.Y,*.U,*.V, 1=*.yuv, 2=*.ppm */
15,17c15,17
<    /* horizontal_size */
<   /* vertical_size */
< 8         /* aspect_ratio_information 1=square pel, 2=4:3, 3=16:9, 4=2.11:1 */
---
> 600   /* horizontal_size */
> 600  /* vertical_size */
> 1         /* aspect_ratio_information 1=square pel, 2=4:3, 3=16:9, 4=2.11:1 */
```

Then convert to ppm, encode and play:

```
for i in g4*jpeg;
  do j=`basename $i .jpeg`; command="convert $i $j.ppm"; echo $command; $command; done
mpeg2encode mpeg2encode.par g4RayTracer.mpg
open g4RayTracer.mpg
```

## 8.11 Debugging geometry with vis

One command in particular - `/vis/drawLogicalVolume` (see below) - is designed to highlight geometry overlaps. But you can explore and check your geometry with simple vis commands. Most examples have a vis.mac file that contains these and several other useful commands — `examples/basic/B1/vis.mac` is a good place to look.

```
/vis/drawVolume
```

will draw the whole detector and your can rotate and zoom. If you want to see just part of the detector:

```
/vis/drawVolume <sub-detector-physical-volume-name> [<copy-number>]
```

draws all volumes with matching name. You may use a regular expression of the form /regexp/, e.g.:

```
/vis/drawVolume /Shape/
```

draws Shape1 and Shape2 (in example B1).

You can accumulate volumes:

```
/vis/drawVolume <physical-volume-name1>
/vis/scene/add/volume <physical-volume-name2>
```

You can also limit the depth of descent of the geometry hierarchy:

```
/vis/drawVolume ! ! 2
```

draws the world to depth 2.

```
/vis/scene/list  # to see what's in the scene.
```

If you are using Qt, what you might find helpful is a little `pick` icon, 5th from the left on the top menu bar of the Qt GUI. It opens a little window and if you click on a shape its properties are displayed in the window.

With a plain OpenGL window, try `/vis/viewer/set/picking`.

If you know the place of the volume in the geometry hierarchy (we call it a "touchable"), you can `/vis/set/touchable` and `/vis/touchable/dump`. Look at the guidance to see how to use the commands. You might find `/vis/drawTree` useful.

All vis commands have extensive guidance. Use `help` or `ls` on the command line or the `Help` tab in Qt, or see "Built-in Commands" in the Application Developers Guide.

### 8.11.1 Using advanced vis tools

`/vis/drawLogicalVolume` can highlight overlaps. But first you may need to identify the offending volumes. *Detecting Overlapping Volumes* describes how to do this.

```
/geometry/test/run
```

gives output such as:

```
Checking overlaps for volume Shape1 ...
-------- WWWW ------- G4Exception-START -------- WWWW -------
*** G4Exception : GeomVol1002
      issued by : G4PVPlacement::CheckOverlaps()
Overlap with volume already placed !
          Overlap is detected for volume Shape1:0
          with Shape2:0 volume's
          local point (0,30,-29.1037), overlapping by at least: 896.261 um
NOTE: Reached maximum fixed number -1- of overlaps reports for this volume !
*** This is just a warning message. ***
-------- WWWW -------- G4Exception-END --------- WWWW -------
```

Pick the offending volume name from the error message above:

```
/vis/touchable/findPath Shape1
```

This gives something like the following output:

```
World 0 Envelope 0 Shape1 0 (mother logical volume: Envelope)
Use this to set a particular touchable with "/vis/set/touchable <path>"
or to see overlaps: "/vis/drawLogicalVolume <mother-logical-volume-name>"
```

Then take the mother logical volume name from the above message:

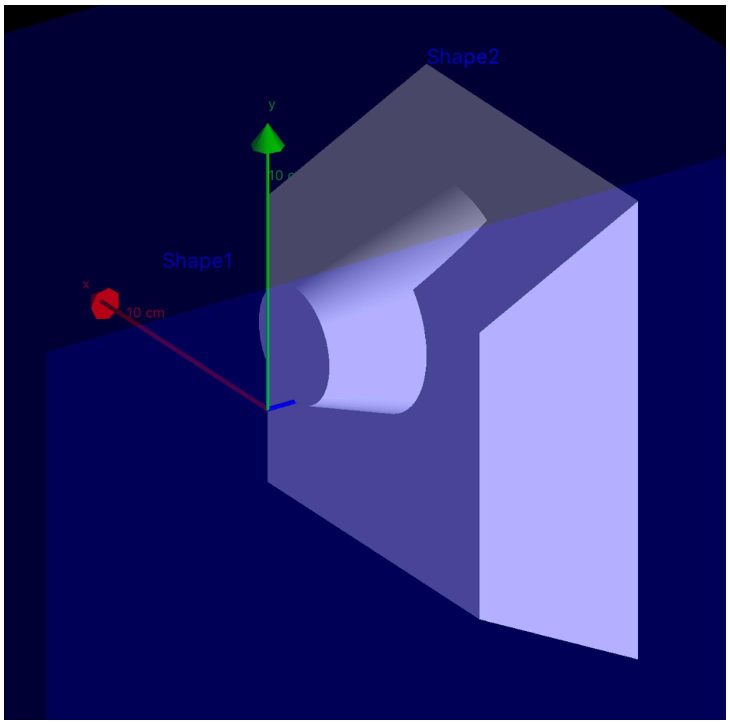


Fig. 8.4: Example B1 with overlapping volumes. (To generate the above one placement in `B1::DetectorConstruction::Construct()` was moved in order to make an overlap.)

```
/vis/drawLogicalVolume Envelope
/vis/viewer/set/style wireframe
```

Now we see the offending volumes highlighted in pink, and the sampling points in pale blue.

It may help to see all the points with:

```
/vis/viewer/set/hiddenMarker false
```

If by chance the offending volumes are "invisible", make them visible with:

```
/vis/viewer/set/culling global false
/vis/viewer/rebuild
```

## 8.12 External Boolean processing

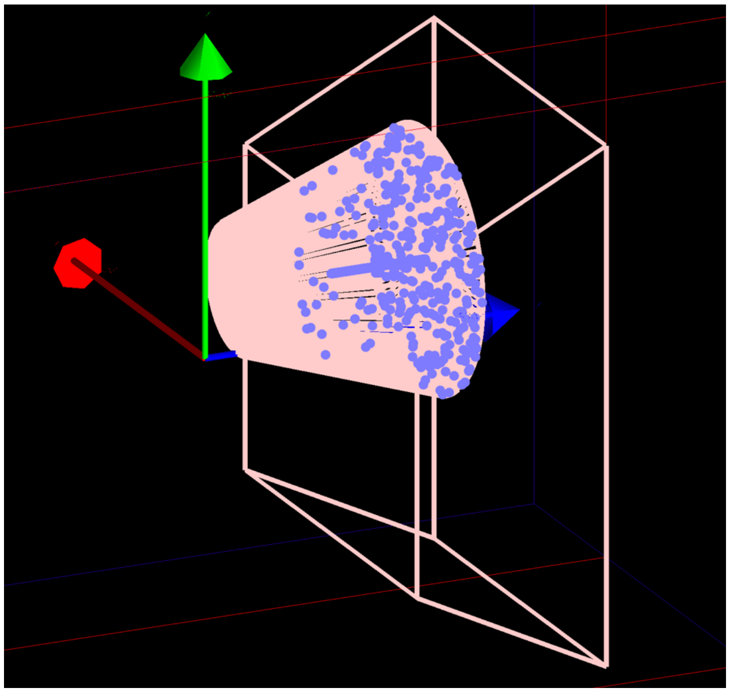


Fig. 8.5: Overlaps highlighted.

CHAPTER

# NINE

# ANALYSIS

## 9.1 Introduction

The analysis category based on g4tools was added in the GEANT4 9.5 release with the aim to provide the users a "light" analysis tool available directly with GEANT4 installation without a need to link their GEANT4 application with an external analysis package. It consists of the analysis manager classes and it includes also the g4tools package.

g4tools provides code to write and read histograms and ntuples in several formats: ROOT, HDF5, XML AIDA format and CSV (comma-separated values format). It is a part of inlib and exlib libraries, that include also other facilities like fitting and plotting.

The output in HDF5 format (since GEANT4 10.4) requires the HDF5 libraries installation as well as GEANT4 libraries built with the -DGEANT4_USE_HDF5=ON CMake option.

The analysis classes provide a uniform, user-friendly interface to g4tools and hide the differences according to a selected output technology from the user. They take care of a higher-level management of the g4tools objects (files, histograms and ntuples), handle allocation and removal of the objects in memory and provide the access methods to them via indexes. They are fully integrated in the GEANT4 framework: they follow GEANT4 coding style and also implement the built-in GEANT4 user interface commands that can be used by users to define or configure their analysis objects.

An example of use of analysis manager classes is provided in basic examples B4 and B5, in their `RunAction` and `EventAction` classes.

## 9.2 Analysis Manager Classes

The analysis manager classes provide uniform interfaces to the g4tools package and hide the differences between use of g4tools classes for the supported output formats (ROOT, HDF5, AIDA XML and CSV).

Since Geant4 10.7, there is a new analysis manager class capable to mix output file formats:

- `G4GenericAnalysisManager`

Mixing of different output types is supported for histogram and profiles objects, only one output type is supported for ntuples.

An analysis manager class is available also for each supported output format:

- `G4CsvAnalysisManager`
- `G4Hdf5AnalysisManager`
- `G4RootAnalysisManager`
- `G4XmlAnalysisManager`

For a simplicity of use, each analysis manager provides the complete access to all interfaced functions though it is implemented via a more complex design. This design allows the user to use all output technologies in an identical way via a generic `G4AnalysisManager` type defined as:

```
using G4AnalysisManager = G4XyzAnalysisManager;
         // where Xyz = Generic, Csv, Hdf5, Root, Xml
```

The managers are implemented as Geant4 singletons. User code will access a pointer to a single instance of the desired manager. The manager is created with the first call to the `Instance()` function and it is deleted by Geant4 kernel at the end of a user application. All objects created via analysis manager are deleted automatically with the manager. In addition to the `G4AnalysisManager` functions, a set of GEANT4 UI commands for creating histograms and setting their properties is implemented in associated messenger classes.

## 9.2.1 Analysis Manager

To use GEANT4 analysis, an instance of the analysis manager must be created. The analysis manager object is created with the first call to `G4AnalysisManager::Instance()`, the next calls to this function will just provide the pointer to this analysis manager object.

The example of the code for creating the analysis manager extracted from the basic B4 example is given below:

```
#include "G4AnalysisManager.hh"

namespace B4
{

RunAction::RunAction()
{
  // Create analysis manager
  auto analysisManager = G4AnalysisManager::Instance();
  analysisManager->SetVerboseLevel(1);
}

}
```

It is recommended, but not necessary, to create the analysis manager in the user run action constructor. This guarantees correct behavior in multi-threading mode.

The G4AnalysisManager.hh header (new since Geant4 11) defines the `G4GenericAnalysisManager` as `G4AnalysisManager` type:

```
#ifndef G4AnalysisManager_h
#define G4AnalysisManager_h

#include "G4GenericAnalysisManager.hh"

using G4AnalysisManager = G4GenericAnalysisManager;

#endif
```

The level of informative printings can be set by `SetVerboseLevel(G4int)`. Currently the levels from 0 (default) up to 4 are supported.

The verbose level can be also set via the UI command:

```
/analysis/verbose level
```

Since Gean4 version 11 the analysis manager is defined as a Geant4 thread-local singleton and users need not and should not delete its instance. Public functions `Clear()` and `Reset()` are provided to allow resetting and deleting all allocated analysis objects and clearing their collections:

```
G4bool Reset();
void Clear();
```

The `Clear()` function can be used as a replacement for deleting the analysis manager in the applications that allocate the analysis data in the begin of run and deleted the analysis manager in the end of run.

Data ressetting can be also performed with a UI command (Since Geant4 11.1):

```
/analysis/reset
```

The list of all analysis objects can be obtained by the `List()` function with an optional boolean argument to choose to display only active objects:

```
G4bool List(G4bool onlyIfActive = true) const;
```

or a UI command:

```
/analysis/list [onlyIfActive]
```

### 9.2.2 Files handling

Below we give an example of opening and closing a file extracted from the basic example B4:

```
#include "G4AnalysisManager.hh"

void RunAction::BeginOfRunAction(const G4Run* run)
{
  // Get analysis manager
  auto analysisManager = G4AnalysisManager::Instance();

  // Open an output file
  G4String fileName = "B4.root";
  analysisManager->OpenFile(fileName);
}

void RunAction::EndOfRunAction(const G4Run* aRun)
{
  // Save histograms
  auto analysisManager = G4AnalysisManager::Instance();
  analysisManager->Write();
  analysisManager->CloseFile();
}
```

The following `G4AnalysisManager` functions are defined for handling files:

```
G4bool OpenFile(const G4String& fileName = "");
G4bool Write();
G4bool CloseFile(G4bool reset = true);
```

The file name can be defined either directly with `OpenFile(const G4String&)` call or separately via `SetFileName(const G4String&)` function before calling `OpenFile()`. It is not possible to change the file name when a file is open and not yet closed. Since Geant4 version 11 multiple files can be hadled at a time, the functions remain valid and define the name of the default file.

The call to `Write()` triggers writing remaining data to all open files and the call to `CloseFile()` closing all files and automatic resetting histograms and ntuples data. Since Geant4 10.5, it is possible to close a file without resetting the histogram data. This feature is newly used in example B5 to keep the histograms available for visualization plotting (resetting of histograms is then performed by a call to `Reset()` in the begin of the next run). Another use case is in MPI extended examples to write the histograms collected on the master rank before they are merged and written in another file.

Since Geant4 11.1, it is possible to call `Write()` multiple times before closing a file. The written objects will then be written with a new name composed of the original name appended with a suffix containing the object cycle number, for example:

```
myHisto   myHisto;1   myHisto;2  etc. in case of the Root output
myHisto   myHisto_v1  myHisto_v2  etc. in case of the other output types
```

The fuctions for file handling can be also performed with UI commands (new since Geant4 11.1):

```
/analysis/openFile [fileName]
/analysis/write
/analysis/closeFile [isReset]
```

A new extended analysis example, AnaEx03, has been added to demonstrate usage of the new analysis commands for file management and in particular writing histograms and ntuples in a file multiple times.

As `G4GenericAnalysisManager` can handle all supported file formats, the file names should be provided with an extension (*.csv*, *.hdf5*, *.root* or *.xml*). To continue using file names without extensions, users can set the default file type using the `G4AnalysisManager` function (new in Geant4 version 11):

```
void SetDefaultFileType(const G4String& value);
```

or a UI command:

```
/analysis/setDefaultFileType fileType
    ... where fileType = csv, hdf5, root, xml
```

## Multiple files handling

Users can choose to write selected objects in a different file than the default one using the `G4AnalysisManager` functions

```
void SetH1FileName(G4int objectId, const G4String& name);
        //... etc. for H2, H3, P1, P2
void SetNtupleFileName(G4int objectId, const G4String& name);
```

The setting can be also performed with UI commands:

```
/analysis/h1/setFileName id name
/analysis/h1/setFileNameToAll name
    ... etc. for h2, h3, p1, p2
/analysis/ntuple/setFileName id fileName
/analysis/ntuple/setFileNameToAll fileName
```

While it is possible to mix output types for histogram and profiles objects, only one output type is supported for ntuples. Saving of histograms and ntuple in two output files in a Root file format is demonstrated in the basic B5 example.

The output specific analysis managers (the analysis manager defined via the `G4XxxAnalysisManager.hh, Xxx = Csv, Hdf5, Root, Xml` include) can also handle multiple files, but all files must be of the same (manager specific) output type.

Depending on the selected output format more files can be generated even when only one file name is set. This is the case of XML, which does not allow writing more than one ntuple in a file, and CSV, which is writing each object (histograms, profile or ntuple) in a separate file. The ntuple (or histogram) file name is then generated automatically from the base file name and the ntuple (or histogram) name.

### File Directories

The file can be optionally structured in sub-directories. Currently only one directory for histograms and/or one directory for ntuples are supported. The directories are created automatically if their names are set to non-empty string values via `G4AnalysisManager` functions:

```
SetHistoDirectoryName(const G4String&)
SetNtupleDirectoryName(const G4String&)
```

The following commands for handling files and directories are available

```
/analysis/setFileName name           # Set name for the output file
/analysis/setHistoDirName name       # Set name for the histograms directory
/analysis/setNtupleDirName name      # Set name for the histograms directory
```

Since Geant4 version 11, in case of the Csv output type, which does not support a directory structure within a file, directories are interpreted as file system directories. The histogram and ntuple files are saved in directories, if their names are set and if the directories exist in the file system. If a directory does not exists (the existence is checked at `SetHisto/NtupleDirectoryName()` call), the histograms/ntuples are written in the current directory and a warning is issued.

## 9.2.3 Histograms

The code for handling histograms given in the following example is extracted the B4 example classes. In this example, the histograms are created in the run action constructor and they are filled in the end of event.

```
#include "G4AnalysisManager.hh"

RunAction::RunAction()
{
  // Create analysis manager
  // ...

  // Creating histograms
  analysisManager->CreateH1("Eabs","Edep in absorber", 100, 0., 800*MeV);
  analysisManager->CreateH1("Egap","Edep in gap", 100, 0., 100*MeV);
}

void EventAction::EndOfEventAction(const G4Run* aRun)
{
  // Fill histograms
  auto analysisManager = G4AnalysisManager::Instance();
  analysisManager->FillH1(0, fEnergyAbs);
  analysisManager->FillH1(1, fEnergyGap);
}
```

### Creating Histograms

A one-dimensional (1D) histogram can be created with one of these two `G4AnalysisManager` functions:

```
G4int CreateH1(const G4String& name, const G4String& title,
               G4int nbins, G4double xmin, G4double xmax,
               const G4String& unitName = "none",
               const G4String& fcnName = "none",
               const G4String& binSchemeName = "linear");

G4int CreateH1(const G4String& name, const G4String& title,
               const std::vector<G4double>& edges,
```

(continues on next page)

(continued from previous page)

```
               const G4String& unitName = "none",
               const G4String& fcnName = "none");
```

where `name` and `title` parameters are self-descriptive. The histogram edges can be defined either via the `nbins`, `xmin` and `xmax` parameters (first function) representing the number of bins, the minimum and maximum histogram values, or via the `const std::vector<G4double>& edges` parameter (second function) representing the edges defined explicitly. The other parameters in both functions are optional and their meaning is explained in *Histograms Properties*.

Two-dimensional (2D) and three-dimensional (3D) histograms can be created with one of these two functions analogous to those for 1D histograms:

```
G4int CreateH2(const G4String& name, const G4String& title,
               G4int nxbins, G4double xmin, G4double xmax,
               G4int nybins, G4double ymin, G4double ymax,
               const G4String& xunitName = "none",
               const G4String& yunitName = "none",
               const G4String& xfcnName = "none",
               const G4String& yfcnName = "none",
               const G4String& xbinScheme = "linear",
               const G4String& ybinScheme = "linear");

G4int CreateH2(const G4String& name, const G4String& title,
               const std::vector<G4double>& xedges,
               const std::vector<G4double>& yedges,
               const G4String& xunitName = "none",
               const G4String& yunitName = "none",
               const G4String& xfcnName = "none",
               const G4String& yfcnName = "none");
```

```
G4int CreateH3(const G4String& name, const G4String& title,
               G4int nxbins, G4double xmin, G4double xmax,
               G4int nybins, G4double ymin, G4double ymax,
               G4int nzbins, G4double zmin, G4double zmax,
               const G4String& xunitName = "none",
               const G4String& yunitName = "none",
               const G4String& zunitName = "none",
               const G4String& xfcnName = "none",
               const G4String& yfcnName = "none",
               const G4String& zfcnName = "none",
               const G4String& xbinSchemeName = "linear",
               const G4String& ybinSchemeName = "linear",
               const G4String& zbinSchemeName = "linear");

G4int CreateH3(const G4String& name, const G4String& title,
               const std::vector<G4double>& xedges,
               const std::vector<G4double>& yedges,
               const std::vector<G4double>& zedges,
               const G4String& xunitName = "none",
               const G4String& yunitName = "none",
               const G4String& zunitName = "none",
               const G4String& xfcnName = "none",
               const G4String& yfcnName = "none",
               const G4String& zfcnName = "none");
```

The meaning of parameters is the same as in the functions for 1D histograms, they are just applied in x, y and z dimensions.

The histograms created with `G4AnalysisManager` get automatically attributed an integer identifier which value is returned from the "Create" function. The default start value is 0 and it is incremented by 1 for each next created histogram. The numbering of 2D and 3D histograms is independent from 1D histograms and so the first created 2D (or 3D) histogram identifier is equal to the start value even when several 1D histograms have been already created.

The start histogram identifier value can be changed either with the `SetFirstHistoId(G4int)` method, which applies the new value to all histogram types, or with the `SetFirstHNId(G4int)`, where `N = 1, 2, 3` methods, which apply the new value only to the relevant histogram type.

All histograms created by `G4AnalysisManager` are automatically deleted with deleting the `G4AnalysisManager` object in the end of the application or by the call to `G4AnalysisManager::Clear()`.

Histograms can be also created via UI commands. The commands to create 1D histogram:

```
/analysis/h1/create                # Create 1D histogram
   name title [nbin min max] [unit] [fcn] [binScheme]
```

The commands to create 2D histogram:

```
/analysis/h2/create                # Create 2D histogram
  name title [nxbin xmin xmax xunit xfcn xbinScheme nybin ymin ymax yunit yfcn yBinScheme]
```

The commands to create 3D histogram:

```
/analysis/h3/create                # Create 3D histogram
  name title [nxbin xmin xmax xunit xfcn xbinScheme nybin ymin ymax
  yunit yfcn yBinScheme nzbin zmin zmax zunit zfcn zBinScheme]
```

### Configuring Histograms

The properties of already created histograms can be changed with use of one of these two functions sets. For 1D histograms:

```
G4bool SetH1(G4int id,
             G4int nbins, G4double xmin, G4double xmax,
             const G4String& unitName = "none",
             const G4String& fcnName = "none",
             const G4String& binSchemeName = "linear");

G4bool SetH1(G4int id,
             const std::vector<G4double>& edges,
             const G4String& unitName = "none",
             const G4String& fcnName = "none");
```

for 2D histograms:

```
G4bool SetH2(G4int id,
             G4int nxbins, G4double xmin, G4double xmax,
             G4int nybins, G4double ymin, G4double ymax,
             const G4String& xunitName = "none",
             const G4String& yunitName = "none",
             const G4String& xfcnName = "none",
             const G4String& yfcnName = "none",
             const G4String& xbinSchemeName = "linear",
             const G4String& ybinSchemeName = "linear");

G4bool SetH2(G4int id,
             const std::vector<G4double>& xedges,
             const std::vector<G4double>& yedges,
             const G4String& xunitName = "none",
             const G4String& yunitName = "none",
             const G4String& xfcnName = "none",
             const G4String& yfcnName = "none");
```

and for 3D histograms:

```
G4bool SetH3(G4int id,
             G4int nxbins, G4double xmin, G4double xmax,
             G4int nzbins, G4double zmin, G4double zmax,
             G4int nybins, G4double ymin, G4double ymax,
             const G4String& xunitName = "none",
             const G4String& yunitName = "none",
             const G4String& zunitName = "none",
             const G4String& xfcnName = "none",
             const G4String& yfcnName = "none",
             const G4String& zfcnName = "none",
             const G4String& xbinSchemeName = "linear",
             const G4String& ybinSchemeName = "linear",
             const G4String& zbinSchemeName = "linear");

G4bool SetH3(G4int id,
             const std::vector<G4double>& xedges,
             const std::vector<G4double>& yedges,
             const std::vector<G4double>& zedges,
             const G4String& xunitName = "none",
             const G4String& yunitName = "none",
             const G4String& zunitName = "none",
             const G4String& xfcnName = "none",
             const G4String& yfcnName = "none",
             const G4String& zfcnName = "none");
```

The histogram is accessed via its integer identifier. The meaning of the other parameters is the same as in “Create” functions.

Histogram properties can be also defined via UI commands. The commands to define 1D histogram

```
/analysis/h1/set id nbin min max [unit] [fcn] [binScheme] # Set parameters
```

The commands to define 2D histogram:

```
# Set parameters for the 2D histogram of #id
/analysis/h2/set
  id nxbin xmin xmax xunit xfcn xbinScheme nybin ymin ymax yunit yfcn yBinScheme

# Set parameters per dimension
/analysis/h2/setX id nbin min max [unit] [fcn] [binScheme] # Set x-parameters
/analysis/h2/setY id nbin min max [unit] [fcn] [binScheme] # Set y-parameters
```

The commands to define 3D histogram:

```
# Set parameters for the 3D histogram of #id
/analysis/h3/set =
  id nxbin xmin xmax xunit xfcn xbinScheme nybin ymin ymax yunit yfcn yBinScheme nzbin zmin zmax␣
↪zunit zfcn zBinScheme

# Set parameters per dimension
/analysis/h3/setX id nbin min max [unit] [fcn] [binScheme] # Set x-parameters
/analysis/h3/setY id nbin min max [unit] [fcn] [binScheme] # Set y-parameters
/analysis/h3/setY id nbin min max [unit] [fcn] [binScheme] # Set z-parameters
```

A limited set of parameters for histograms plotting, the histogram and the histogram axis titles, can be also defined via functions

```
G4bool SetH1Title(G4int id, const G4String& title);
G4bool SetH1XAxisTitle(G4int id, const G4String& title);
G4bool SetH1YAxisTitle(G4int id, const G4String& title);
//
G4bool SetH2Title(G4int id, const G4String& title);
G4bool SetH2XAxisTitle(G4int id, const G4String& title);
G4bool SetH2YAxisTitle(G4int id, const G4String& title);
```

(continues on next page)

(continued from previous page)

```
G4bool SetH2ZAxisTitle(G4int id, const G4String& title);
//
G4bool SetH3Title(G4int id, const G4String& title);
G4bool SetH3XAxisTitle(G4int id, const G4String& title);
G4bool SetH3YAxisTitle(G4int id, const G4String& title);
G4bool SetH3ZAxisTitle(G4int id, const G4String& title);
```

The corresponding UI commands

```
/analysis/h1/setTitle id title    # Set title for the 1D histogram of #id
/analysis/h1/setXaxis id title    # Set x-axis title for the 1D histogram
/analysis/h1/setYaxis id title    # Set y-axis title for the 1D histogram
```

The same set of commands is available for the other histogram types and profiles, under the appropriate directory.

### Filling Histograms

The histogram values can be filled using the functions:

```
G4bool FillH1(G4int id, G4double value,
              G4double weight = 1.0);
G4bool FillH2(G4int id, G4double xvalue, G4double yvalue,
              G4double weight = 1.0);
G4bool FillH3(G4int id,
              G4double xvalue, G4double yvalue, G4double zvalue,
              G4double weight = 1.0);
```

where the weight can be given optionally.

The histograms can be also scaled with a given factor using the functions:

```
G4bool ScaleH1(G4int id, G4double factor);
G4bool ScaleH2(G4int id, G4double factor);
G4bool ScaleH3(G4int id, G4double factor);
```

### Histograms Properties

The following properties, additional to those defined in g4tools, can be added to histograms via `G4AnalysisManager`:

- *Unit*: if a histogram is defined with a unit, all filled values are automatically converted to this defined unit and the unit is added to the histogram axis title.
- *Function*: if a histogram is defined with a function, the function is automatically executed on the filled values and its name is added to the histogram axis title. When a histogram is defined with both unit and function the unit is applied first. The available functions: `log`, `log10`, `exp`.
- *Binning scheme*: user can select logarithmic binning scheme besides the linear one (default). The available binning schemes: `linear`, `log`.
- *Activation*: see the *Activation of Analysis Objects* section.
- *ASCII option*: if activated the histogram is also printed in an ASCII file when Write() function is called.
- *Plotting option*: if activated the histogram is plotted in a file of Postscript format when Write() function is called. See more details in the *Plotting* section.
- *File name*: if defined, the histogram is written in a new file with the histogram file name which is open at the `Write()` call and then closed with all files open at `CloseFile()` call.

### 9.2.4 Profiles

Profile histograms (profiles) are used to display the mean value of Y and its error for each bin in X. The displayed error is by default the standard error on the mean (i.e. the standard deviation divided by the sqrt(n).) An example of use of 1D profiles can be found in `extended/electromagnetic/TestEm2`. Though the functions for creating and manipulating profiles are very similar to those for histograms, they are described in this section.

#### Creating Profiles

A one-dimensional (1D) profile can be created with one of these two `G4AnalysisManager` functions

```
G4int CreateP1(const G4String& name, const G4String& title,
               G4int nbins, G4double xmin, G4double xmax,
               G4double ymin = 0, G4double ymax = 0,
               const G4String& xunitName = "none",
               const G4String& yunitName = "none",
               const G4String& xfcnName = "none",
               const G4String& yfcnName = "none",
               const G4String& xbinSchemeName = "linear");

G4int CreateP1(const G4String& name, const G4String& title,
               const std::vector<G4double>& edges,
               G4double ymin = 0, G4double ymax = 0,
               const G4String& xunitName = "none",
               const G4String& yunitName = "none",
               const G4String& xfcnName = "none",
               const G4String& yfcnName = "none");
```

where `name` and `title` parameters are self-descriptive. The profile edges can be defined either via the `nbins`, `xmin` and `xmax` parameters (first function) representing the number of bins, the minimum and maximum profile values, or via the `const std::vector<G4double>& edges` parameter (second function) representing the edges defined explicitly. If `ymin` and `ymax` parameters are provides, only values between these limits will be considered at filling time. The other parameters in both functions are optional and their meaning is explained in *Profiles Properties*.

A two-dimensional (2D) profile can be created with one of these two functions analogous to those for 1D profiles:

```
G4int CreateP2(const G4String& name, const G4String& title,
               G4int nxbins, G4double xmin, G4double xmax,
               G4int nybins, G4double ymin, G4double ymax,
               G4double zmin = 0, G4double zmax = 0,
               const G4String& xunitName = "none",
               const G4String& yunitName = "none",
               const G4String& zunitName = "none",
               const G4String& xfcnName = "none",
               const G4String& yfcnName = "none",
               const G4String& zfcnName = "none",
               const G4String& xbinSchemeName = "linear",
               const G4String& ybinSchemeName = "linear");

G4int CreateP2(const G4String& name, const G4String& title,
               const std::vector<G4double>& xedges,
               const std::vector<G4double>& yedges,
               G4double zmin = 0, G4double zmax = 0,
               const G4String& xunitName = "none",
               const G4String& yunitName = "none",
               const G4String& zunitName = "none",
               const G4String& xfcnName = "none",
               const G4String& yfcnName = "none",
               const G4String& zfcnName = "none");
```

The meaning of parameters is the same as in the functions for 1D profiles, they are just applied in x, y and z dimensions.

The profiles created with `G4AnalysisManager` get automatically attributed an integer identifier which value is returned from the "Create" function. The default start value is 0 and it is incremented by 1 for each next created profile. The numbering of 2D profiles is independent from 1D profiles and so the first created 2D profile identifier is equal to the start value even when several 1D profiles have been already created.

The start profile identifier value can be changed either with the `SetFirstProfileId(G4int)` method, which applies the new value to both 1D and 2D profile types, or with the `SetFirstPNId(G4int)`, where `N = 1, 2` methods, which apply the new value only to the relevant profile type.

All profiles created by `G4AnalysisManager` are automatically deleted with deleting the `G4AnalysisManager` object in the end of the application or by the call to `G4AnalysisManager::Clear()`.

Profiles can be also created via UI commands. The commands to create 1D profile

```
/analysis/p1/create                # Create 1D profile
  name title [nxbin xmin xmax xunit xfcn xbinScheme ymin ymax yunit yfcn]
```

The commands to create 2D profile:

```
/analysis/p2/create                # Create 2D profile
  name title [nxbin xmin xmax xunit xfcn xbinScheme nybin ymin ymax yunit yfcn yBinScheme zmin␣
↪zmax zunit zfcn]
```

## Configuring Profiles

The properties of already created profiles can be changed with use of one of these two functions sets. For 1D profiles

```
G4bool SetP1(G4int id,
             G4int nbins, G4double xmin, G4double xmax,
             G4double ymin = 0, G4double ymax = 0,
             const G4String& xunitName = "none",
             const G4String& yunitName = "none",
             const G4String& xfcnName = "none",
             const G4String& yfcnName = "none",
             const G4String& xbinSchemeName = "linear");

G4bool SetP1(G4int id,
             const std::vector<G4double>& edges,
             G4double ymin = 0, G4double ymax = 0,
             const G4String& xunitName = "none",
             const G4String& yunitName = "none",
             const G4String& xfcnName = "none",
             const G4String& yfcnName = "none");
```

and for 2D profiles:

```
G4bool SetP2(G4int id,
             G4int nxbins, G4double xmin, G4double xmax,
             G4int nybins, G4double ymin, G4double ymax,
             G4double zmin = 0, G4double zmax = 0,
             const G4String& xunitName = "none",
             const G4String& yunitName = "none",
             const G4String& zunitName = "none",
             const G4String& xfcnName = "none",
             const G4String& yfcnName = "none",
             const G4String& zfcnName = "none",
             const G4String& xbinSchemeName = "linear",
             const G4String& ybinSchemeName = "linear");

G4bool SetP2(G4int id,
             const std::vector<G4double>& xedges,
             const std::vector<G4double>& yedges,
```

(continues on next page)

(continued from previous page)

```
              G4double zmin = 0, G4double zmax = 0,
              const G4String& xunitName = "none",
              const G4String& yunitName = "none",
              const G4String& zunitName = "none",
              const G4String& xfcnName = "none",
              const G4String& yfcnName = "none",
              const G4String& zfcnName = "none");
```

The profile is accessed via its integer identifier. The meaning of the other parameters is the same as in "Create" functions.

Profiles properties can be also defined via UI commands. The commands to define 1D profile

```
/analysis/p1/set                  # Set parameters for the 1D histogram of #id
  id nxbin xmin xmax xunit xfcn xbinScheme ymin ymax yunit yfcn
```

The commands to create or define 2D profile:

```
/analysis/p2/set                  # Set parameters for the 2D profile of #id
  id nxbin xmin xmax xunit xfcn xbinScheme nybin ymin ymax yunit yfcn yBinScheme zmin zmax zunit␣
↪zfcn
```

A limited set of parameters for profiles plotting, the profile and the profile axis titles, can be also defined via functions

```
G4bool SetP1Title(G4int id, const G4String& title);
G4bool SetP1XAxisTitle(G4int id, const G4String& title);
G4bool SetP1YAxisTitle(G4int id, const G4String& title);
//
G4bool SetP2Title(G4int id, const G4String& title);
G4bool SetP2XAxisTitle(G4int id, const G4String& title);
G4bool SetP2YAxisTitle(G4int id, const G4String& title);
G4bool SetP2ZAxisTitle(G4int id, const G4String& title);
```

The parameters can be also set via the same set of UI commands as the histogram parameters available under the appropriate directory.

### Filling Profiles

The profile values can be filled using the functions:

```
G4bool FillP1(G4int id,
              G4double xvalue, G4double yvalue,
              G4double weight = 1.0);
G4bool FillP2(G4int id,
              G4double xvalue, G4double yvalue, G4double zvalue,
              G4double weight = 1.0);
```

where the weight can be given optionally.

The profiles can be also scaled with a given factor using the functions:

```
G4bool ScaleP1(G4int id, G4double factor);
G4bool ScaleP2(G4int id, G4double factor);
```

### Profiles Properties

All histogram features described in the *Histograms Properties* section are also available for profiles.

## 9.2.5 Ntuples

In the following example the code for handling ntuples extracted from basic example B4, from the `B4::RunAction` and `B4a::EventAction` classes, is presented.

```
#include "G4AnalysisManager.hh"

RunAction::RunAction()
 : G4UserRunAction()
{
  // Create analysis manager
  // ...

  // Create ntuple
  man->CreateNtuple("B4", "Edep and TrackL");
  man->CreateNtupleDColumn("Eabs");
  man->CreateNtupleDColumn("Egap");
  man->FinishNtuple();
}

void EventAction::EndOfEventAction(const G4Run* aRun)
{
  G4AnalysisManager* man = G4AnalysisManager::Instance();
  man->FillNtupleDColumn(0, fEnergyAbs);
  man->FillNtupleDColumn(1, fEnergyGap);
  man->AddNtupleRow();
}
```

Since 10.0 release, there is no limitation for the number of ntuples that can be handled by `G4AnalysisManager`. Handling of two ntuples is demonstrated in extended analysis/AnaEx01 example.

### Creating Ntuples

An ntuple can be created using the following set of functions:

```
G4int CreateNtuple(const G4String& name, const G4String& title);

// Create columns in the last created ntuple
G4int CreateNtupleXColumn(const G4String& name);
void  FinishNtuple();

// Create columns in the ntuple with given id
G4int CreateNtupleXColumn(G4int ntupleId, const G4String& name);
void  FinishNtuple(G4int ntupleId);
```

The first set is demonstrated in the example. The columns can take the values of `G4int`, `G4float`, `G4double` or `G4String` type which is also reflected in the `CreateNtupleXColumn()` function names. where `X` can be `I, F, D or S`.

It is also possible to define ntuple columns of `std::vector` of `G4int`, `G4float` or `G4double` values using the functions:

```
// Create columns of vector in the last created ntuple
G4int CreateNtupleXColumn(
        const G4String& name, std::vector<Xtype>& vector);

```

(continues on next page)

(continued from previous page)

```
// Create columns of vector in the ntuple with given id
G4int CreateNtupleXColumn(G4int ntupleId,
        const G4String& name, std::vector<Xtype>& vector);
```

where `[X, Xtype]` can be `[I, G4int]`, `[F, G4float]` or `[D, G4double]`.

When all ntuple columns are created, the ntuple has to be closed using `FinishNtuple()` function.

The ntuples created with `G4AnalysisManager` get automatically attributed an integer identifier which value is returned from the "Create" function. The default start value is 0 and it is incremented by 1 for each next created ntuple. The start ntuple identifier value can be changed with the `SetFirstNtupleId(G4int)` function.

The integer identifiers are also attributed to the ntuple columns. The numbering of ntuple columns is independent for each ntuple, the identifier default start value is 0 and it is incremented by 1 for each next created column regardless its type (*I*, *F*, *D* or *S*). (If the third ntuple column of a different type than `double` (`int` or `float`) is created in the demonstrated example, its identifier will have the value equal 2.) The start ntuple column identifier value can be changed with the `SetFirstNtupleColumnId(G4int)` function.

When calls to `CreateNtuple-Column()` and `FinishNtuple()` succeed the call to `CreateNtuple()`, the `ntupleId` argument need not to be specified even when creating several ntuples. However this order is not enforced and the second set of functions with `ntupleId` argument is provided to allow the user to create the ntuples and their columns in whatever order.

All ntuples and ntuple columns created by `G4AnalysisManager` are automatically deleted with deleting the `G4AnalysisManager` object in the end of the application or by the call to `G4AnalysisManager::Clear()`.

Since Geant4 11.2 ntuple can be also created via UI commands:

```
/analysis/ntuple/create name title
/analysis/ntuple/create[I|F|D|S]Column name
/analysis/ntuple/finish
```

## Filling Ntuples

The ntuple values of fundamental and string types can be filled using the functions:

```
// Methods for ntuple with id = FirstNtupleId
G4bool FillNtupleIColumn(G4int id, G4int value);
G4bool FillNtupleFColumn(G4int id, G4float value);
G4bool FillNtupleDColumn(G4int id, G4double value);
G4bool FillNtupleSColumn(G4int id, const G4String& value);

// Methods for ntuple with id > FirstNtupleId (when more ntuples exist)
G4bool FillNtupleIColumn(G4int ntupleId, G4int columnId, G4int value);
G4bool FillNtupleFColumn(G4int ntupleId, G4int columnId, G4float value);
G4bool FillNtupleDColumn(G4int ntupleId, G4int columnId, G4double value);
G4bool FillNtupleSColumn(G4int ntupleId, G4int id, const G4String& value);
```

If only one ntuple is defined in the user application, the ntuple identifier, `ntupleId`, need not to be specified and the first set can be used. The second set of functions has to be used otherwise.

The `FillNtupleXColumn()` functions should not be called for the columns of vector type, as these are directly associated with the vector reference and so don't need to be filled.

When all ntuple columns are filled, the ntuple fill has to be closed by calling `AddNtupleRow()`:

```
// Methods for ntuple with id = FirstNtupleId
G4bool AddNtupleRow();
```

(continues on next page)

(continued from previous page)

```
// Methods for ntuple with id > FirstNtupleId (when more ntuples exist)
G4bool AddNtupleRow(G4int ntupleId);
```

### 9.2.6 Analysis objects handling

While the functions for creating and filling analysis objects (histograms, profiles and ntuples) are specific to the object type, the functions for objects access, activation and newly added objects deleting have the same signature for all objects and can be described in a common section.

#### Accessing Analysis Objects

`G4AnalysisManager` provides the direct access to the g4tools histogram, profiles and ntuple objects via their integer identifiers:

```
tools::histo::h1d* GetH1(G4int id, G4bool warn = true, G4bool onlyIfActive = true) const;
tools::histo::h2d* GetH2(G4int id, G4bool warn = true, G4bool onlyIfActive = true) const;
tools::histo::h3d* GetH3(G4int id, G4bool warn = true, G4bool onlyIfActive = true) const;
tools::histo::p1d* GetP1(G4int id, G4bool warn = true, G4bool onlyIfActive = true) const;
tools::histo::p2d* GetP2(G4int id, G4bool warn = true, G4bool onlyIfActive = true) const;

ntuple_type GetNtuple(G4bool warn = true, G4bool onlyIfActive = true) const;
ntuple_type GetNtuple(G4int id, G4bool warn = true, G4bool onlyIfActive = true) const;
```

If an object with a given `id` is not found, a warning is issued unless it is explicitly disabled by the user. The `id` parameter can be ommitted in `GetNtuple`, the analysis manager then returns the first ntuple created. The last optional boolean argument can be used to choose to get only an active object.

The `GetNtuple` return type depends on the choice of the analysis manager type. `G4GenericAnalysisManager` (default) returns `tools::ntuple_booking*`, that is used to construct ntuples and is common to all managers. The output specific managers return their specific ntuple type.

With using `auto`, users need not to declare the g4tools object type explicitly. In example B4, the g4tools histogram functions `mean()` and `rms()` are called:

```
auto analysisManager = G4AnalysisManager::Instance();
if ( analysisManager->GetH1(1) ) {
  G4cout << "\n ----> print histograms statistic \n" << G4endl;
  G4cout << " EAbs : mean = " << analysisManager->GetH1(1)->mean()
         << " rms = " << analysisManager->GetH1(1)->rms(),
         << G4endl;
  // ...
}
```

Besides the fast access to analysis objects via their integer identifiers, histograms and profiles can be also accessed by their names using the `G4AnalysisManager` function providing the conversion from a name in a histogram identifier:

```
G4int GetH1Id(const G4String& name, G4bool warn = true) const;
         // etc for H2, H3, P1, P2
```

This way is however less efficient and it is not recommended for frequently called functions as e.g. `Fill()`.

The list of defined analysis objects can be obtained by the `List*()` function with an optional boolean argument to choose to display only active objects:

```
G4bool ListH1(G4bool onlyIfActive = true) const;
         // etc for H2, H3, P1, P2, Ntuple
```

or by corresponding UI commands:

```
/analysis/h1/list [onlyIfActive]
          // etc for h2, h3, p1, p2, ntuple
```

### Activation of Analysis Objects

The activation option allows the user to activate only selected histograms, profiles or ntuples. When this option is activated, only the analysis objects marked as activated are returned, filled or saved in a file. This feature is intensively used in extended/electromagnetic examples where all histograms are first created inactivated:

```
auto analysisManager = G4AnalysisManager::Instance();
analysisManager->SetActivation(true);
// define histogram parameters name, title, nbins, vmin, vmax
G4int id = analysisManager->CreateH1(name, title, nbins, vmin, vmax);
analysisManager->SetH1Activation(id, false);
```

and then selected histograms are activated in macros, using the analysis "set" command

```
/analysis/h1/set 1 100 0   50 cm        #track length of primary
/analysis/h1/set 2 100 0  300 none      #nb steps of primary
```

The activation option is not switched on by default. It has to be activated either via analysisManager `SetActivation(true)` call as above or via the UI command:

```
/analysis/setActivation true|false   # Set activation option
```

When no parameters need to be changed a histogram can be activated using "setActivation" command:

```
/analysis/h1/setActivation id true|false     # Set activation to histogram #id
/analysis/h1/setActivationToAll  true|false  # Set activation to all 1D histograms.
         // etc for h2, h3, p1, p2, ntuple
```

### Deleting Selected Analysis Objects

Though there is no need for users to take care of memory management of the analysis objects (histograms, profiles and ntuples) as this is completely handled by the analysis manager, since Geant4 11.2, users have a possibility to delete selected objects using the following analysis manager functions:

```
G4bool DeleteH1(G4int id, G4bool keepSetting = false);
         // etc for H2, H3, P1, P2, Ntuple
```

and corresponding UI commands:

```
/analysis/h1/delete id [keepSetting]
         // etc for h2, h3, p1, p2, ntuple
```

The integer identifiers of already created objects do not change with deleting selected objects. `GetHn|Pn|Ntuple(idx)` function will return `nullptr` if the object at the `idx` position was deleted. It is under user responsibility to test the returned value before its use. When an analysis manager function is called with `idx` of an object that was deleted, a warning is issued, but no segmentation violation error happens.

When a new object is created after some objects were deleted, it gets attributed the first available identifier. If, for example, we create five `h1` histograms before `Run 0` and then delete the `h1` histograms with `id = 2` and `id = 4` before `Run 1`, and create again two new `h1` histograms before `Run 2` these new histograms get `id = 2` and `id = 4`.

If `keepSetting` was set to `true`, the additional information (object activation, object file name etc.) is kept after object deleting and it is automatically set to the new object created at this `id`.

Deleting and re-creating of selected histograms is demonstrated in extended analysis example, AnaEx03.

### 9.2.7 Plotting

For the interactive histograms and profiles plotting available with the Geant4 visualization system see *Visualization of histograms (plotting)*.

In this section we describe how a graphics output file in the Postscript format containing selected histograms and profiles can be produced with analysis tools (since GEANT4 10.2). The batch plotting can be activated using `G4AnalysisManager` functions:

```
auto analysisManager = G4AnalysisManager::Instance();
analysisManager->SetH1Plotting(id, true);
   // etc for H2, H3, P1, P2
```

or using the UI commands

```
/analysis/h1/setPlotting id true|false     # (In)Activate plottig for 1D histogram #id
/analysis/h1/setPlottingToAll true|false   # (In)Activate plotting for all 1D histograms.
       # etc. for h2, h3, p1, p2
```

If GEANT4 libraries are built with support for Freetype font rendering, user can choose from three plotting styles:

- ROOT_default: ROOT style with high resolution fonts (default)
- hippodraw: hippodraw style with high resolution fonts
- inlib_default: PAW style with low resolution fonts")

otherwise only the `inlib_default` style with low resolution fonts is available.

The page size of the graphics output is fixed to A4 format. Users can choose the page layout which is defined by the number columns and the number of rows in a page. Depending on the selected plotting style, the maximum number of plots is limited to 3 columns x 5 rows for the styles with high resolution fonts and to 2 columns x 3 rows for the `inlib_default` style.

Finally, users can also customize the plot dimensions, which represent the plotter window size (width and height) in pixels.

The customization of the plotting can be done via the UI commands in `/analysis/plot` directory:

```
/analysis/plot/setStyle  styleName
/analysis/plot/setLayout  columns rows
/analysis/plot/setDimensions  width height
```

Opening more configuration parameters for users customisation can be considered in future according to the users feedback.

### 9.2.8 Parallel Processing

As well as all other GEANT4 categories, the analysis code has been adapted for multi-threading. In multi-threading mode, the analysis manager instances are internally created on the master and thread workers and data accounting is processed in parallel on workers threads.

Histograms produced on thread workers are automatically merged on `Write()` call and the result is written in a master file. Merging is protected by a mutex locking, using `G4AutoLock` utility.

Ntuples produced on thread workers are, by default, written on separate files, which names are generated automatically from a base file name, a thread identifier and eventually also an ntuple name. Since GEANT4 version 10.3 it is possible to activate merging of ntuples with the ROOT output type:

```
auto analysisManager = G4AnalysisManager::Instance();
analysisManager->SetNtupleMerging(true);
```

The ntuples produced on workers will be then progressively being merged to the main ntuples on the master.

By default, the ntuples are written at the same file as the final histograms. Users can also select merging in a given number of files via the optional parameter of the `SetNtupleMerging()` function:

```
void SetNtupleMerging(G4bool mergeNtuples, G4int nofReducedNtupleFiles = 0);
```

No merging of ntuples is provided with HDF5, CSV and AIDA XML formats.

Users can override the defaults and change the ntuple merging mode with a new function (since 10.5, with the added second argument in 10.6):

```
void SetNtupleRowWise(G4bool rowWise, G4bool rowMode = true);
```

The available merging modes:

- *column-wise not preserving rows* (`rowWise = false, rowMode=false`)
  The fastest option, but without preserving the ntuple rows after merging.
- *row-wise* (`rowWise = true, rowMode value is not used`)
  In order to preserve an "event point of view" (the ntuple rows) after merging, a row-wise merging mode was introduced in 10.4 and became a default in 10.5. In this mode, columns are defined as leaves of a single TBranch attached to each ntuple per worker.
  This approach has an inconvenience that as column-wise is used in sequential mode and row-wise is used in parallel, the user will have different data schema (different organizations of TBranches and TLeaves) in files, which may complicate his life when reading back his data. While this does not affect a simple analysis using `TTree::Draw("branchName")`, different methods need to be used to access the data per branch. The example macros with a simple analysis are provided in basic examples (for example B4/macros/plotNtuple.C) and a complete access to data is demonstrated in several extended examples (for example medical/dna/dnaphysics/plot.C).
- *column-wise with preserving rows* (`rowWise = false, rowMode=true`)
  The column-wise mode enhanced with preserving the ntuple rows, it became a default in 10.6. It requires larger memory size than the other two modes depending on the users applications.

Users can also change the default values of basket size (32000) and basket entries (4000) using new functions (since 10.6).

```
void SetBasketSize(unsigned int basketSize);
void SetBasketEntries(unsigned int basketEntries);
```

In previous Geant4 versions the basket size could be set as an optional argument of the `SetNtupleMerging` function.

To simplify the scaling of a GEANT4 application across nodes on a cluster GEANT4 provides the support of MPI. In particular it is possible to run a hybrid MPI/MT application that uses MPI to scale across nodes and MT to scale across cores. This is demonstrated in the extended example `parallel/MPI/exMPI03` which includes usage of GEANT4 analysis for histograms.

A new example `parallel/MPI/exMPI04`, the same as exMPI03 with added ntuple, shows how to merge, using g4tools, ntuples via MPI in sequential mode, so that the entire statistics is accumulated in a single output file. If MT is enabled, the ntuples are merged from threads to files per ranks. Combined MT + MPI merging is not yet supported.

### 9.2.9 Supported Features and Limitations

The analysis category based on g4tools is provided with certain limitations that can be reduced according to the feedback from GEANT4 users and developers.

Below is a summary of currently supported features in Root, Hdf5, Csv and Xml manager classes:

- Histogram types: 1D, 2D, 3D of `double`
- Profile types: 1D, 2D of `double`
- Ntuple column types: `int`, `float`, `double`, `G4String`, `std::vector<int>`, `std::vector<float>`, `std::vector<double>`, `std::vector<std::string>`
- Optional directory structure limited to one directory for histograms and/or one for ntuples

## 9.3 Analysis Reader Classes

The analysis reader classes allow to read in g4analysis objects from the files generated by the analysis manager(s) during processing GEANT4 application.

An analysis reader class is available for each supported output format:

- `G4CsvAnalysisReader`
- `G4Hdf5AnalysisReader`
- `G4RootAnalysisReader`
- `G4XmlAnalysisReader`

For a simplicity of use, each analysis reader provides the complete access to all interfaced functions though it is implemented via a more complex design. This design allows the user to use all output technologies in an identical way via a generic `G4AnalysisReder` type defined as:

- **`using G4AnalysisReder = G4XyzAnalysisReader;`** where `Xyz = Csv, Hdf5, Root, Xml`

The readers are implemented as Geant4 singletons. User code will access a pointer to a single instance of the desired reader object. The reader is created with the first call to the `Instance()` function and it is deleted by Geant4 kernel at the end of a user application. All objects created via analysis reader are deleted automatically with the manager.

While the histograms and profiles objects handled by the analysis reader are of the same type as those handled by the analysis manager, the reader's ntuple type is different.

All objects read with `G4AnalysisReader` (histograms, profiles and ntuples) get automatically attributed an integer identifier which value is returned from the "Read" or "GetNtuple" function. The default start value is 0 and it is incremented by 1 for each next created object. The numbering each object type is independent from other objects types and also from the numbering of the same object type in analysis manager. The start identifier value can be changed in the same way as with the analysis manager (see *Creating Histograms*).

The read objects can be accessed in the analysis reader via their integer identifiers or by their names in the same way as in the analysis manager (see *Accessing Analysis Objects*). Note that the type of read ntuple is different from the ntuple type in the analysis manager.

The specific manager classes are singletons and so it is not possible to create more than one instance of an analysis reader of one type, e.g. `G4RootAnalysisReader`. However two analysis reader objects of different types can coexist.

As well as all other GEANT4 categories, the analysis code has been adapted for multi-threading. In multi-threading mode, the analysis reader instances are internally created on the master or thread workers, depending on the client code call, and data reading can be processed in parallel on workers threads.

### 9.3.1 Analysis Reader

For reading in the output files created with `G4AnalysisManager`, an instance of the analysis reader must be created. The analysis reader object is created with the first call to `G4AnalysisReader::Instance()` the next calls to this function will just provide the pointer to this analysis manager object.

The example of the code for creating the analysis reader for the Root output type is given below:

```
#include "G4RootAnalysisReader.hh"

using G4AnalysisReader = G4RootAnalysisReader;

// Create (or get) analysis reader
auto analysisReader = G4AnalysisReader::Instance();
analysisReader->SetVerboseLevel(1);
```

The `using` declaration defines the `G4RootAnalysisReader` as `G4AnalysisReader` type.

The level of informative printings can be set by `SetVerboseLevel(G4int)`. Currently the levels from 0 (default) up to 4 are supported.

### 9.3.2 File handling

The name of file to be read can be specified either via `G4AnalysisReader::SetFileName()` function, or directly when reading an object. It is possible to change the base file name at any time. The analysis reader can handle more than one file at same time.

```
auto analysisReader = G4AnalysisReader::Instance();
// Define a base file name
analysisReader->SetFileName("MyFileName");
```

The following functions are defined for handling files:

```
void SetFileName(const G4String& fileName);
G4String GetFileName() const;
```

A file is open only when any "Read" function is called. When more objects are read from the same file (Xml, Root), the file is open only once. When reading an object without specifying the file name explicitly in "Read" call, the object is searched in all open files in the order of their creation time.

### 9.3.3 Histograms and Profiles

In the following example the code for reading an histogram is presented.

```
// Code to create (or get) analysis reader
auto analysisReader = G4AnalysisReader::Instance();

// Define a base file name
analysisReader->SetFileName("MyFileName");

// Read 1D histogram of "Edep" name
G4int h1Id = analysisReader->ReadH1("Edep");
if ( h1Id >= 0 ) {
  G4H1* h1 = analysisReader->GetH1(h1Id);
  if ( h1 ) {
    G4cout << "   H1: "
           << "   mean: " << h1->mean() << " rms: " << h1->rms() << G4endl;
  }
}
```

The histograms and profiles can be read with these `G4AnalysisReader` functions:

```
G4int ReadH1(const G4String& h1Name, const G4String& fileName = "");
G4int ReadH2(const G4String& h2Name, const G4String& fileName = "");
G4int ReadH3(const G4String& h3Name, const G4String& fileName = "");
G4int ReadP1(const G4String& h1Name, const G4String& fileName = "");
G4int ReadP2(const G4String& h2Name, const G4String& fileName = "");
```

where `hNname` is the name of the object to be read from a file. The file name can be defined explicitly for each reading object.

All histograms and profiles created by `G4AnalysisReader` are automatically deleted with deleting the `G4AnalysisReader` object in the end of the application.

### 9.3.4 Ntuples

In the following example the code for reading ntuples is presented.

```
// Code to create (or get) analysis reader
auto analysisReader = G4AnalysisReader::Instance();

// Define a base file name
analysisReader->SetFileName("MyFileName");

// Read ntuple
G4int ntupleId = analysisReader->GetNtuple("TrackL");;
if ( ntupleId >= 0 ) {
  G4double trackL;
  analysisReader->SetNtupleDColumn("Labs", trackL);
  G4cout << "Ntuple TrackL, reading selected column Labs" << G4endl;
  while ( analysisReader->GetNtupleRow() ) {
      G4cout << counter++ << "th entry: "
             << "  TrackL: " << trackL << std::endl;
  }
}
```

When the ntuple columns are associated with the variables of the appropriate type, the ntuple they can be read in a loop with `GetNtupleRow()` function. The function returns true until all data are read in.

An overview of all available functions for ntuple reading is given below:

```
// Methods to read ntuple from a file
G4int GetNtuple(const G4String& ntupleName, const G4String& fileName = "");

// Methods for ntuple with id = FirstNtupleId
G4bool SetNtupleXColumn(const G4String& columnName, Xtype& value);
G4bool SetNtupleXColumn(const G4String& columnName, std::vector<Xtype>& vector);
G4bool GetNtupleRow();

// Methods for ntuple with id > FirstNtupleId
G4bool SetNtupleXColumn(G4int ntupleId,
                        const G4String& columnName, Xtype& value);
G4bool SetNtupleXColumn(G4int ntupleId,
                        const G4String& columnName, std::vector<Xtype>& vector);
G4bool GetNtupleRow(G4int ntupleId);
```

where `[X, Xtype]` in `SetNtupleXColumn()` can be `[I, G4int]`, `[F, G4float]`, `[D, G4double]` or `[S, G4String]`.

All ntuples and ntuple columns created by `G4AnalysisReader` are automatically deleted with deleting the `G4AnalysisReader` object in the end of the application.

## 9.4 Accumulables

The classes for users accumulables management were added in 10.2 release for the purpose of simplification of users application code. The accumulables objects are named variables registered to the accumulable manager, which provides the access to them by name and performs their merging in multi-threading mode according to their defined merge mode. Their usage is demonstrated in the basic examples B1 and B3a.

To better reflect the meaning of these objects, the classes base name "Parameter" used in 10.2 was changed in "Accumulable" in 10.3. Further integration in the GEANT4 framework is foreseen in the next GEANT4 versions.

### 9.4.1 G4Accumulable<T>

`G4Accumulable<T>` templated class can be used instead of built-in types in order to facilitate merging of the values accumulated on workers to the master thread. The `G4Accumulable<T>` object has, besides its value of the templated type `T`, also a name, the initial value, which the value is set to in `Reset()` function and a merge mode, specifying the operation which is performed in `Merge()` function.

The accumulable object can be either instantiated using its constructor and registered in `G4AccumulablesManager` explicitly, or it can be created using `G4AccumulablesManager::CreateAccumulable()` function, their registering is then automatic. The first way is used in the basic examples B1 and B3a:

```
// B1/include/RunAction.hh
class RunAction : public G4UserRunAction
{
  // ...
  private:
    G4Accumulable<G4double> fEdep  = 0.;
    G4Accumulable<G4double> fEdep2 = 0.;
};

// B1/src/RunAction.cc
RunAction::RunAction()
{
  // ..
  // Register accumulable to the accumulable manager
  G4AccumulableManager* accumulableManager = G4AccumulableManager::Instance();
```

(continues on next page)

(continued from previous page)

```
  accumulableManager->RegisterAccumulable(fEdep);
  accumulableManager->RegisterAccumulable(fEdep2);
}
```

An alternative way of creating an accumulable using `G4AccumulablesManager` is demonstrated below:

```
// B1/src/RunAction.cc
RunAction::RunAction()
{
  // ..
  // Accumulables can be also created via accumulable manager
  G4AccumulableManager* accumulableManager = G4AccumulableManager::Instance();
  accumulableManager->CreateAccumulable<G4double>("EdepBis", 0.);
  accumulableManager->CreateAccumulable<G4double>("Edep2Bis", 0.);
}
```

The `G4AccumulablesManager` takes ownership of the accumulables created by its `CreateAccumulable()` function the accumulables allocated in the user code has to be deleted in the user code.

Since GEANT4 10.3, the name of the accumulable can be omitted. A generic name “accumulable_N”, where N is the current number of registered objects, will be then attributed.

In multi-threading mode all accumulables registered to `G4AccumulablesManager` accumulated on workers can be merged to the master thread by calling `G4AccumulablesManager::Merge()` function. This step may be not necessary in future after a planned closer integration of G4Accumulable classes in the GEANT4 kernel.

```
// B1/src/RunAction.cc
void RunAction::EndOfRunAction(const G4Run* run)
{
  // ...
  // Merge accumulables
  G4AccumulableManager* accumulableManager = G4AccumulableManager::Instance();
  accumulableManager->Merge();
}
```

The merging mode can be specified using the third (or the second one, if the name is omitted) `G4Accumulable<T>` constructor argument. The merge modes are defined in `G4MergeMode` class enumeration:

```
enum class G4MergeMode {
  kAddition,        // "Or" if boolean type
  kMultiplication,  // "And" if boolean type
  kMaximum,         // "Or" if boolean type
  kMinimum          // "And" if boolean type
};
```

The default accumulable merge operation is addition.

The registered accumulables can be accessed via `G4AccumulablesManager` by name or by the id, attributed in the order of registering:

```
// ...
G4AccumulableManager* accumulableManager = G4AccumulableManager::Instance();
// Access accumulables by name
G4double edepBis  = accumulableManager->GetAccumulable<G4double>("EdepBis")->GetValue();
G4double edep2Bis = accumulableManager->GetAccumulable<G4double>("Edep2Bis")->GetValue();

// Access accumulables by id
G4VAccumulable* accumulable = accumulableManager->GetAccumulable(id);
```

### 9.4.2 User defined accumulables

Users can define their own accumulable class derived from `G4VAccumulable` abstract base class. An example of a `ProcessCounterAccumulable` class, implementing an accumulable holding a map of the processes occurrences by the processes names, is given below. Such processes occurrences map is used in several electromagnetic extended examples, e.g. TestEm1.

ProcCounterAccumulable.hh:

```
#include "G4VAccumulable.hh"
#include "globals.hh"
#include <map>
class ProcCounterAccumulable : public G4VAccumulable
{
  public:
    ProcCounterAccumulable(const G4String& name)
      : G4VAccumulable(name, 0), fProcCounter() {}
    virtual ~ProcCounterAccumulable() {}

    void CountProcesses(G4String procName);

    virtual void Merge(const G4VAccumulable& other);
    virtual void Reset();

  private:
    std::map<G4String,G4int> fProcCounter;
};
```

ProcCounterAccumulable.cc:

```
void ProcCounterAccumulable::Merge(const G4VAccumulable& other)
{
  const ProcCounterAccumulable& otherProcCounterAccumulable
    = static_cast<const ProcCounterAccumulable&>(other);

  std::map<G4String,G4int>::const_iterator it;
  for (it = otherProcCounterAccumulable.fProcCounter.begin();
       it != otherProcCounterAccumulable.fProcCounter.end(); ++it) {

    G4String procName = it->first;
    G4int otherCount  = it->second;
    if ( fProcCounter.find(procName) == fProcCounter.end()) {
      fProcCounter[procName] = otherCount;
    }
    else {
      fProcCounter[procName] += otherCount;
    }
  }
}

void ProcCounterAccumulable::Reset()
{
  fProcCounter.clear();
}
```

The implementation of the `CountProcesses()` function is identical as in `Run::CountProcesses()` function in TestEm1.

## 9.5 g4tools

g4tools is a "namespace protected" part of inlib and exlib which is of some interest for GEANT4, mainly the histograms, the ntuples and the code to write them at the ROOT, HDF5, AIDA XML and CSV file formats. The idea of g4tools is to cover, with a very light and easy to install package, what is needed to do analysis in a "GEANT4 batch program".

As g4tools is distributed through GEANT4 and in order to avoid potential namespace clashes with other codes that use the inlib/exlib to do GEANT4 visualization (as for the g4view application or some of the exlib examples), the *inlib* and *exlib* namespaces had been automatically changed to *tools* in the g4tools distribution. Since in principle GEANT4 users will not have to deal directly with the g4tools classes, but will manipulate histograms and ntuples through the G4AnalysisManager, we are not going to extensively document the g4tools classes here. Interested people are encouraged to go at the inlib/exlib web pages for that (see inlib/exlib site).

### 9.5.1 g4tools package

#### g4tools code is pure header

As explained in inlib/exlib, the code found in g4tools is "pure header". This comes from the need to have an easy way to build applications, as the ioda one, from smartphone, passing by tablets and up to various desktops (UNIX and Windows). For example, if building an application targeted to the Apple AppStore and GooglePlay, the simplest way is to pass through Xcode and the Android make system (or Eclipse), and having not to build libraries simplifies a lot the handling of all these IDEs for the same application. A fallback of that is that the installation of g4tools (if not using the one coming with GEANT4) is straightforward, you simply unzip the file containing the source code! To build an application using g4tools, as for inlib/exlib, you simply have to declare to your build system the "-I" toward the unfolded directory and do "Build and Run".

#### g4tools test

g4tools comes with test programs of its own that may be useful in case of problems (for example porting on a not yet covered platform). You can build and run them with:

```
UNIX> <get g4tools.zip>
UNIX> <unzip g4tools.zip>
UNIX> cd g4tools/test/cpp
UNIX> ./build
UNIX> ./tools_test_histo
UNIX> ./tools_test_wroot
UNIX> etc...
```

and on Windows:

```
DOS> <setup VisualC++ so that CL.exe is in your PATH>
DOS> <get g4tools.zip>
DOS> <unzip g4tools.zip> (you can use the unzip.exe of CYGWIN)
DOS> cd g4tools\test\cpp
DOS> .\build.bat
DOS> .\tools_test_histo.exe
DOS> .\tools_test_wroot.exe
DOS> etc...
```

### g4tools in Geant4

The `g4tools` header files are distributed in the Geant4 source in the `source/analysis/include/tools` directory and in the Geant4 installation, they are installed in `include/tools` directory. The `g4tools` test programs, included only in Geant4 development versions, can be downloaded with the `g4tools-[version].zip` file from the `inexlib` development site).

While the Geant4 analysis manager provides the methods for booking and filling the g4tools objects, it does not interface all public functions. Users can access the g4tools objects (see *Accessing Analysis Objects*) and use the g4tools API described in the next section to get the needed informations.

## 9.5.2 User API

We describe here some of the public methods potentially seen by a user doing analysis.

### Booking and filling

```
h1d(const std::string& title,unsigned int Xnumber,double Xmin,double Xmax);
h1d(const std::string& title,const std::vector<double>& edges);

bool fill(double X,double Weight = 1);
```

example

```
#include <tools/histo/h1d>
#include <tools/randd>
...
tools::histo::h1d h("Gauss",100,-5,5);
tools::rgaussd rg(1,2);
for(unsigned int count=0;count<entries;count++) h.fill(rg.shoot(),1.4);
```

### Mean and rms

```
tools::histo::h1d h("Gauss",100,-5,5);
...
std::cout << " mean " << h.mean() << ", rms " << h.rms() << std::endl;
```

### Bin infos

When doing a:

```
bool fill(double X,double Weight = 1);
```

the histogram class maintains, for each bin, the number of entries, the sum of weights that we can note “Sw”, the sum of W by W “Sw2”, the sum of X by Weight “Sxw”, the sum of X by X by W “Sx2w”. Then bin method names reflect these notations, for example to get the 50 bin sum of X*X*W:

```
double Sx2w = h.bin_Sx2w(50);
```

and the same for the other sums:

```
double Sw = h.bin_Sw(50);
double Sw2 = h.bin_Sw2(50);
double Sxw = h.bin_Sxw(50);
unsigned int n = h.bin_entries(50);
```

You can have also all infos on all bins with:

```
tools::histo::h1d h(...);
...
const std::vector<unsigned int>& _entries = h.bins_entries();
const std::vector<double>& _bins_sum_w = h.bins_sum_w();
const std::vector<double>& _bins_sum_w2 = h.bins_sum_w2();
const std::vector< std::vector<double> >& _bins_sum_xw = h.bins_sum_xw();
const std::vector< std::vector<double> >& _bins_sum_x2w = h.bins_sum_x2w();
```

for example to dump bin 50 of an histo booked with 100 bins:

```
std::cout << "entries[50] = " << _entries[50] << std::endl;
std::cout << "  sum_w[50] = " << _bins_sum_w[50] << std::endl;
std::cout << " sum_w2[50] = " << _bins_sum_w2[50] << std::endl;
std::cout << " sum_xw[50] = " << _bins_sum_xw[50][0] << std::endl;   //0 = xaxis
std::cout << "sum_x2w[50] = " << _bins_sum_x2w[50][0] << std::endl;  //0 = xaxis
```

(Take care that the [0] entries in the upper vectors are for the "underflow bin" and the last one is for the "overflow bin").

### All data

You can get all internal data of an histo through the histo_data class:

```
const tools::histo::h1d::hd_t& hdata = h.dac();  //dac=data access.
```

and then, for example, find back the bins infos with:

```
const std::vector<unsigned int>& _entries = hdata.m_bin_entries;
const std::vector<double>& _bins_sum_w = hdata.m_bin_Sw;
const std::vector<double>& _bins_sum_w2 = hdata.m_bin_Sw2;
const std::vector< std::vector<double> >& _bins_sum_xw = hdata.m_bin_Sxw;
const std::vector< std::vector<double> >& _bins_sum_x2w = hdata.m_bin_Sx2w;
// dump bin 50 :
std::cout << "entries[50] = " << _entries[50] << std::endl;
std::cout << "  sum_w[50] = " << _bins_sum_w[50] << std::endl;
std::cout << " sum_w2[50] = " << _bins_sum_w2[50] << std::endl;
std::cout << " sum_xw[50] = " << _bins_sum_xw[50][0] << std::endl;   //0 = xaxis
std::cout << "sum_x2w[50] = " << _bins_sum_x2w[50][0] << std::endl;  //0 = xaxis
```

See the tools/histo/histo_data class for all internal fields.

### Projections

From a 2D histo, you can get the x projection with:

```
tools::histo::h1d* projection = tools::histo::projection_x(h2d,"ProjX");
...
delete projection;
```

See test/cpp/histo.cpp for example code. Other slicing and projection methods are:

```
// h2d -> h1d. (User gets ownership of the returned object).
h1d* slice_x(const h2d&,int y_beg_ibin,int y_end_ibin,const std::string& title);
h1d* projection_x(const h2d&,const std::string& title);
h1d* slice_y(const h2d&,int x_beg_ibin,int x_end_ibin,const std::string& title);
h1d* projection_y(const h2d&,const std::string& title);
// h2d -> p1d. (User gets ownership of the returned object).
p1d* profile_x(const h2d&,int y_beg_ibin,int y_end_ibin,const std::string& title);
```

(continues on next page)

(continued from previous page)

```
p1d* profile_x(const h2d&,const std::string&);
p1d* profile_y(const h2d&,int x_beg_ibin,int x_end_ibin,const std::string& title);
p1d* profile_y(const h2d&,const std::string& title);
// h3d -> h2d. (User gets ownership of the returned object).
h2d* slice_xy(const h3d&,int z_beg_ibin,int z_end_ibin,const std::string& title);
h2d* projection_xy(const h3d&,const std::string& title);
h2d* slice_yz(const h3d&,int x_beg_ibin,int x_end_ibin,const std::string& title);
h2d* projection_yz(const h3d&,const std::string& title);
h2d* slice_xz(const h3d&,int y_beg_ibin,int y_end_ibin,const std::string& title);
h2d* projection_xz(const h3d&,const std::string& title);
```

CHAPTER

TEN

# EXAMPLES

## 10.1 Introduction

The GEANT4 toolkit includes several fully coded examples that demonstrate the implementation of the user classes required to build a customized simulation.

The new "basic" examples cover the most typical use-cases of a GEANT4 application while keeping simplicity and ease of use. They are provided as a starting point for new GEANT4 application developers.

A set of "extended" examples range from the simulation of a non-interacting particle and a trivial detector to the simulation of electromagnetic and hadronic physics processes in a complex detector. Some of these examples require some libraries in addition to those of GEANT4.

The "advanced" examples cover the use-cases typical of a "toolkit"-oriented kind of development, where real complete applications for different simulation studies are provided.

All examples can be compiled and run without modification. Most of them can be run both in interactive and batch mode using the input macro files (`*.in`) and reference output files (`*.out`) provided. Most examples are run routinely as part of the validation, or testing, of official releases of the GEANT4 toolkit.

The previous set of examples oriented to novice users, "novice", has been refactored in "basic" and "extended" examples sets in GEANT4 10.0. The information about the original set of these examples can be found at the last section of this chapter.

## 10.2 Basic Examples

### 10.2.1 Basic Examples Summary

Descriptions of the 5 basic examples are provided here along with links to source code documentation automatically generated with Doxygen.

*Example B1* (see also Doxygen page)

- Simple geometry with a few solids
- Geometry with simple placements (G4PVPlacement)
- Scoring total dose in a selected volume in user action classes
- Using `G4Accumulable` for automatic merging of scored values inmulti-threading mode
- GEANT4 physics list (QBBC)

*Example B2* (see also Doxygen page)

- Simplified tracker geometry with uniform magnetic field
- Geometry with simple placements (G4PVPlacement) and parameterisation (G4PVParameterisation)
- Scoring within tracker via G4 sensitive detector and hits

- GEANT4 physics list (FTFP_BERT) with step limiter
- Started from novice N02 example

*Example B3* (see also Doxygen page)

- Schematic Positron Emission Tomography system
- Geometry with simple placements with rotation (G4PVPlacement)
- Radioactive source
- Scoring within Crystals via G4 scorers plus via user actions (a), via user own Run object (b)
- Using `G4Accumulable` for automatic merging of scored values in multi-threading mode (a) and `G4StatAnalysis` for accumulating statistics (b)
- Modular physics list built via builders provided in GEANT4

*Example B4* (see also Doxygen page)

- Simplified calorimeter with layers of two materials
- Geometry with replica (G4PVReplica)
- Scoring within layers in four ways: via user actions (a), via user own Run object (b), via G4 sensitive detector and hits (c) and via scorers (d)
- GEANT4 physics list (FTFP_BERT)
- Saving histograms and ntuple in a file using GEANT4 analysis tools
- UI commands defined using G4GenericMessenger
- Started from novice/N03 example

*Example B5* (see also Doxygen page)

- A double-arm spectrometer with wire chambers, hodoscopes and calorimeters with a local constant magnetic field
- Geometry with placements with rotation, replicas and parameterisation
- Scoring within wire chambers, hodoscopes and calorimeters via G4 sensitive detector and hits
- GEANT4 physics list (FTFP_BERT) with step limiter
- UI commands defined using G4GenericMessenger
- Saving histograms and ntuple in two files using GEANT4 analysis tools
- Plotting of histograms with visualization drivers
- Started from extended/analysis/A01

The next three tables display the “item charts” for the examples currently prepared in the basic level. (Table 10.1, Table 10.2, and Table 10.3.)

Table 10.1: The “item chart” for basic level examples *B1* and *B2*.

| | *Example B1* | *Example B2* |
|---|---|---|
| Description | Simple application for accounting dose in a selected volume | Fixed target tracker geometry |
| Geometry | • solids: box, cons, trd<br>• simple placements with translation | • solids: box, tubs<br>• simple placements with translation (a)<br>• parameterised volume (b)<br>• uniform magnetic field |
| Physics | GEANT4 physics list: QBBC | GEANT4 physics list: FTFP_BERT |
| Primary generator | Particle gun | Particle gun |
| Scoring | User action classes | Sensitive detector & hits |
| Vis/GUI | Detector & trajectory drawing | • Detector, trajectory & hits drawing<br>• GUI |
| Stacking | | |
| Analysis | | |

Table 10.2: The “item chart” for basic level examples *B3* and *B4*.

| | *Example B3* | *Example B4* |
|---|---|---|
| Description | Schematic Positron Emitted Tomography system | Simplified calorimeter with layers of two materials |
| Geometry | • solids: box, tubs<br>• simple placements with rotation | • solids: box<br>• simple placements with translation<br>• replica<br>• uniform magnetic field |
| Physics | Modular physics list with GEANT4 builders | GEANT4 physics list: FTFP_BERT |
| Primary generator | Radioactive source (particle gun with Fluor ions) | Particle gun |
| Scoring | Multi functional (sensitive) detector & scorers and<br>• User action classes<br>• User own run object | • User action classes<br>• User own object (runData)<br>• Sensitive detector & hits<br>• Multi functional (sensitive) detector & scorers |
| Vis/GUI | Detector, trajectory & hits drawing | • Detector, trajectory & hits drawing<br>• GUI |
| Stacking | Killing all neutrina | |
| Analysis | | Histograms 1D, ntuple |

Table 10.3: The “item chart” for basic level example *B5*.

| | *Example B5* |
|---|---|
| Description | Double-arm spectrometer with several detectors and a local constant magnetic field |
| Geometry | • solids: box, tubs<br>• simple placements with rotation<br>• replica<br>• parameterised volume<br>• local constant magnetic field<br>• modifying geometry between runs |
| Physics | GEANT4 physics list: FTFP_BERT |
| Primary generator | Particle gun |
| Scoring | Sensitive detectors & hits |
| Vis/GUI | • Detector, trajectory & hits drawing<br>• User defined visualization attributes<br>• Plotting of histograms |
| Stacking | |
| Analysis | • Histograms 1D, ntuple<br>• Saving two files per run |

## 10.2.2 Basic Examples Macros

All basic examples can be run either interactively or in a batch mode (see section *How to Define the main() Program* and *How to Execute a Program*) and they are provided with the following set of macros:

- `init_vis.mac`
- `vis.mac`
- `[gui.mac]`
- `[plotter.mac]`
- `run1.mac`, `run2.mac`
- `exampleBN.in`

The selection is done automatically according to the application build configuration.

The `init_vis.mac` macro is always executed just after the GEANT4 kernel and user application classes instantiation. It sets first some defaults, then performs GEANT4 kernel initialization and finally calls the `vis.mac` macro with visualization setting.

The `vis.mac` macros in each of the examples all have the same structure - except for example B1, see below. There are only a few lines in each example with a setting different from the other examples and so they can be easily spotted when looking in the macro. Various commands are proposed in commented blocks of lines with explanations so that a user can just uncomment lines and observe the effect. Additionally, in example B4, there are some visualization tutorial macros in `macros/visTutor/`. See more on visualization in section *How to Visualize the Detector and Events* and chapter *Visualization*.

From Release 9.6 the `vis.mac` macro in example B1 has additional commands that demonstrate additional functionality of the vis system, such as displaying text, axes, scales, date, logo and shows how to change viewpoint and style. Consider copying these to your favourite example or application. To see even more commands use `help` or `ls` or browse the available UI commands in section *Built-in Commands*.

The `gui.mac` macros are provided in examples B2, B4 and B5. This macro is automatically executed if GEANT4 is built with any GUI session. See more on graphical user interfaces in section *How to Set Up an Interactive Session*.

When running interactively, the example program stops after processing the `init_vis.mac` macro and the GEANT4 kernel initialization, invoked from the macro, with the prompt `Idle>`. At this stage users can type in the commands from `run1.mac` line by line (recommended when running the example for the first time) or execute all commands at once using the "`/control/execute run1.mac`" command.

The `run2.mac` macros define conditions for execution a run with a larger number of events and so they are recommended to be executed in a batch. The `exampleBN.in` macros are also supposed to be run in a batch mode and their outputs from the GEANT4 system testing are available in the files `exampleBN.out`.

The `plotter.mac` macro is provided in example B5. This macro shows how to use the plotting coming with some of the visualization drivers (for example the ToolsSG ones) to see histograms. This macro can be run interactively, after example start at the prompt `Idle>`, using the "`/control/execute plotter.mac`" command, the content of the histograms is then displayed at the end of each run.

## 10.2.3 Multi-threading

### Multi-threading mode

All basic examples have been migrated to multi-threading (MT). No special steps are needed to build the examples in multi-threading mode. They will automatically run in MT when they are built against the GEANT4 libraries built with MT mode activated, otherwise they will run in sequential mode.

The choice of multi-threading mode is handled automatically by the use of `G4RunManagerFactory` in the example `main()`:

```
#include "G4RunManagerFactory.hh"

// ...

auto* runManager = G4RunManagerFactory::Create();
```

The concrete type of `runManager` will be `G4MTRunManager` when the Geant4 install used supports multithreading, or *G4RunManager* ` otherwise.

### `Action Initialization` class [B1, B2, B3, B4, B5]

See B1, B2, B3, B4, B5

A newly introduced `Bn::ActionInitialization` class derived from `G4VUserActionInitialization`, present in all basic examples, instantiates and registers all user action classes with the GEANT4 kernel.

While in sequential mode the action classes are instantiated just once, via invocation of the method `BnActionInitialization::Build()`. In multi-threading mode the same method is invoked for each worker thread, so all user action classes are defined thread-locally.

A run action class is instantiated both thread-locally and globally; that is why its instance is created also in the method `Bn::ActionInitialization::BuildForMaster()`, which is invoked only in multi-threading mode.

### 10.2.4 Example B1

See also Doxygen page

**Basic concept:**

This example demonstrates a simple (medical) application within which users will familiarize themselves with simple placement, use the NIST material database, and can utilize electromagnetic and/or hadronic physics processes. Two items of information are collected in this example: the energy deposited and the total dose for a selected volume.

This example uses the GEANT4 physics list QBBC, which is instantiated in the main() function. It requires data files for electromagnetic and hadronic processes. See more on installation of the datasets in Geant4 Installation Guide - Geant4 Build Options. The following datasets: G4LEDATA, G4LEVELGAMMADATA, G4NEUTRONXSDATA, G4SAIDXSDATA and G4ENSDFSTATEDATA are mandatory for this example.

**Namespace:**

Since Geant4 version 11 all example classes are defined in namespace B1.

**Classes:**

**`B1::DetectorConstruction`** The geometry is constructed in the `B1::DetectorConstruction` class. The setup consists of a box shaped envelope containing two volumes: a circular cone and a trapezoid.
Some common materials from medical applications are used. The envelope is made of water and the two inner volumes are made from tissue and bone materials. These materials are created using the `G4NistManager` class, which allows one to build a material from the NIST database using their names. Available materials and their compositions can be found in the Appendix *Material Database*.
The physical volumes are made from Constructive Solid Geometry (CSG) solids and placed without rotation using the `G4PVPlacement` class.

**`B1::PrimaryGeneratorAction`** The default kinematics is a 6 MeV gamma, randomly distributed in front of the envelope across 80% of the transverse (X,Y) plane. This default setting can be changed via the commands of the `G4ParticleGun` class.

**`B1::SteppingAction`** It is in the `UserSteppingAction()` function that the energy deposition is collected for a selected volume.

**`B1::EventAction`** The statistical event by event accumulation of energy deposition. At the end of event, the accumulated values are passed in `B1::RunAction` and summed over the whole run.

**`B1::RunAction`** Sums the event energy depositions. In multi-threading mode the energy deposition accumulated in `G4Accumulable` objects per worker is merged to the master. Information about the primary particle is printed in this class along with the computation of the dose. An example of creating and computing new units (e.g., dose) is also shown in the class constructor.
`G4Accumulable<G4double>` type instead of `G4double` is used for the `B1::RunAction` data members in order to facilitate merging of the values accumulated on workers to the master. At present the accumulables have to be registered to `G4AccumulablesManager` and `G4ParametersManager::Merge()` has to be called from the users code. This is planned to be further simplified with a closer integration of `G4Accumulable` classes in the GEANT4 kernel next year.

### 10.2.5 Example B2

See also Doxygen page

This example simulates a simplified fixed target experiment. To demonstrate alternative ways of constructing the geometry two variants are provided: B2a (explicit construction) and B2b (parametrized volumes).

The set of available particles and their physics processes are defined in the FTFP_BERT physics list. This GEANT4 physics list is instantiated in the main() function. It requires data files for electromagnetic and hadronic processes. See more on installation of the datasets in |Geant4| Installation Guide. The following datasets: G4LEDATA, G4LEVELGAMMADATA, G4NEUTRONXSDATA, G4SAIDXSDATA and G4ENSDFSTATEDATA are mandatory for this example.

This example also illustrates how to introduce tracking constraints like maximum step length via `G4StepLimiter`, and minimum kinetic energy, etc., via the `G4UserSpecialCuts` processes. This is accomplished by adding `G4StepLimiterPhysics` to the physics list.

**Namespaces:**

Since Geant4 version 11 the example classes are defined in namespaces B2, B2a and B2b

**Classes:**

**`B2[a, b]::DetectorConstruction`** The setup consists of a target followed by six chambers of increasing transverse size at defined distances from the target. These chambers are located in a region called the Tracker region. Their shape are cylinders constructed as simple cylinders (in `B2a::DetectorConstruction`) and as parametrised volumes (in `B2b::DetectorConstruction`) - see also B2b::ChamberParameterisation class.
In addition, a global uniform transverse magnetic field can be applied using `G4GlobalMagFieldMessenger`, instantiated in `ConstructSDandField()` with a non zero field value, or via an interactive command. An instance of the `B2::TrackerSD` class is created and associated with each logical chamber volume (in B2a) and with the one `G4LogicalVolume` associated with `G4PVParameterised` (in B2b).
One can change the materials of the target and the chambers interactively via the commands defined in `B2a::DetectorMessenger` (or `B2b::DetectorMessenger`).
This example also illustrates how to introduce tracking constraints like maximum step length, minimum kinetic energy etc. via the G4UserLimits class and associated G4StepLimiter and G4UserSpecialCuts processes. The maximum step limit in the tracker region can be set by the interactive command defined in `B2a::DetectorMessenger` (or `B2b::DetectorMessenger`).

**`B2::PrimaryGeneratorAction`** The primary generator action class employs the `G4ParticleGun`. The primary kinematics consists of a single particle which hits the target perpendicular to the entrance face. The type of the particle and its energy can be changed via the G4 built-in commands of the `G4ParticleGun` class.

**`B2::EventAction`** The event number is written to the log file every requested number of events in `BeginOfEventAction()` and `EndOfEventAction()`. Moreover, for the first 100 events and every 100 events thereafter information about the number of stored trajectories in the event is printed as well as the number of hits stored in the `G4VHitsCollection`.

**`B2::RunAction`** The run number is printed at `BeginOfRunAction()`, where the `G4RunManager` is also informed how to `SetRandomNumberStore` for storing initial random number seeds per run or per event.

**`B2::TrackerHit`** The tracker hit class is derived from `G4VHit`. In this example, a tracker hit is a step by step record of the track identifier, the chamber number, the total energy deposit in this step, and the position of the energy deposit.

**`B2::TrackerSD`** The tracker sensitive detector class is derived from `G4VSensitiveDetector`. In `ProcessHits()` - called from the GEANT4 kernel at each step - it creates one hit in the selected volume so long as energy is deposited in the medium during that step. This hit is inserted in a HitsCollection. The HitsCollection is printed at the end of each event (via the method `B2::TrackerSD::EndOfEvent()`), under the control of the "/hits/verbose 2" command.

### 10.2.6 Example B3

See also Doxygen page

This example simulates a Schematic Positron Emission Tomography system. To demonstrate alternative ways of accumulation event statistics in a run two variants are provided: B3a (using new `G4Accumulable` class) and B3b (using `G4Run` class).

**Namespaces:**

Since Geant4 version 11 the example classes are defined in namespaces B3, B3a and B3b

**Classes:**

**B3::DetectorConstruction** Crystals are circularly arranged to form a ring. A number rings make up the full detector (gamma camera). This is done by positioning Crystals in Ring with an appropriate rotation matrix. Several copies of Ring are then placed in the full detector.

The Crystal material, Lu2SiO5, is not included in the G4Nist database. Therefore, it is explicitly built in `DefineMaterials()`.

Crystals are defined as scorers in `DetectorConstruction::CreateScorers()`. There are two G4MultiFunctionalDetector objects: one for the Crystal (EnergyDeposit), and one for the Patient (DoseDeposit).

**B3::PhysicsList** The physics list contains standard electromagnetic processes and the radioactiveDecay module for GenericIon. It is defined in the `B3::PhysicsList` class as a GEANT4 modular physics list with registered GEANT4 physics builders:

- `G4DecayPhysics`
- `G4RadioactiveDecayPhysics`
- `G4EmStandardPhysics`

**B3::PrimaryGeneratorAction** The default particle beam is an ion (F18), at rest, randomly distributed within a zone inside a patient and is defined in `GeneratePrimaries()`.

**B3a::EventAction, B3a::RunAction** Energy deposited in crystals is summed by `G4Scorer`.

The scorers hits are saved in form of ntuples in a Root file using Geant4 analysis tools. This feature is activated in the main () function with instantiating G4TScoreNtupleWriter.

At the end of event, the values accumulated in `B3aEventAction` are passed in `B3a::RunAction` and summed over the whole run. In multi-threading mode the data accumulated in `G4Accumulable` objects per workers is merged to the master in `B3a::RunAction::EndOfRunAction()` and the final result is printed on the screen.

`G4Accumulable<>` type instead of `G4double` and `G4int` types is used for the `B3a::RunAction` data members in order to facilitate merging of the values accumulated on workers to the master. At present the accumulables have to be registered to `G4AccumulablesManager` and `G4AccumulablesManager::Merge()` has to be called from the users code. This is planned to be further simplified with a closer integration of `G4Accumulable` classes in the GEANT4 kernel next year.

**B3b::Run, B3b::RunAction** Energy deposited in crystals is summed by `G4Scorer`. `B3b::Run::RecordEvent()` collects information event by event from the hits collections, and accumulates statistics for `B3b::RunAction::EndOfRunAction()`. In multi-threading mode the statistics accumulated per worker is merged to the master in `Run::Merge()`.

In addition, results for dose are accumulated in a standard floating-point summation and using a new lightweight statistical class called `G4StatAnalysis`. The `G4StatAnalysis` class records four values: (1) the sum, (2) sum^2, (3) number of entries, and (4) the number of entries less than mean * machine-epsilon (the machine epsilon is the difference between 1.0 and the next value representable by the floating-point type). From these 4 values, `G4StatAnalysis` provides the mean, FOM, relative error, standard deviation, variance, coefficient of variation, efficiency, r2int, and r2eff.

**B3::StackingAction** Beta decay of Fluorine generates a neutrino. One wishes not to track this neutrino; therefore one kills it immediately, before created particles are put in a stack.

### 10.2.7 Example B4

See also Doxygen page

This example simulates a simple Sampling Calorimeter setup. To demonstrate several possible ways of data scoring, the example is provided in four variants: B4a, B4b, B4c, B4d. (See also examples/extended/electromagnetic/TestEm3).

The set of available particles and their physics processes are defined in the FTFP_BERT physics list. This GEANT4 physics list is instantiated in the main() function. It requires data files for electromagnetic and hadronic processes. See more on installation of the datasets in |Geant4| Installation Guide. The following datasets: G4LEDATA, G4LEVELGAMMADATA, G4NEUTRONXSDATA, G4SAIDXSDATA and G4ENSDFSTATEDATA are mandatory for this example.

**Namespaces:**

Since Geant4 version 11 the example classes are defined in namespaces B4, B4a, B4b, B4c and B4d

**Classes:**

**`B4[c, d]::DetectorConstruction`** The calorimeter is a box made of a given number of layers. A layer consists of an absorber plate and of a detection gap. The layer is replicated. In addition a transverse uniform magnetic field can be applied using `G4GlobalMagFieldMessenger`, instantiated in `ConstructSDandField()` with a non zero field value, or via interactive commands.

**`B4::PrimaryGeneratorAction`** The primary generator action class uses `G4ParticleGun`. It defines a single particle which hits the calorimeter perpendicular to the input face. The type of the particle can be changed via the G4 built-in commands of the `G4ParticleGun` class.

**`B4::RunAction`** It accumulates statistics and computes dispersion of the energy deposit and track lengths of charged particles with the aid of analysis tools. H1D histograms are created in `BeginOfRunAction()` for the energy deposit and track length in both Absorber and Gap volumes. The same values are also saved in an ntuple. The histograms and ntuple are saved in the output file in a format according to a selected file extension. In `EndOfRunAction()`, the accumulated statistics and computed dispersion are printed. When running in multi-threading mode, the histograms accumulated on threads are automatically merged in a single output file, while the ntuple is written in files per thread.

**Classes in B4a (scoring via user actions):**

**`B4a::SteppingAction`** In `UserSteppingAction()` the energy deposit and track lengths of charged particles in each step in the Absorber and Gap layers are collected and subsequently recorded in `B4a::EventAction`.

**`B4a::EventAction`** It defines data members to hold the energy deposit and track lengths of charged particles in the Absorber and Gap layers. In `EndOfEventAction()`, these quantities are printed and filled in H1D histograms and ntuple to accumulate statistic and compute dispersion.

**Classes in B4b (via user own object):**

**`B4b::RunData`** A data class, derived from `G4Run`, which defines data members to hold the energy deposit and track lengths of charged particles in the Absorber and Gap layers. It is instantiated in `B4b::RunAction::GenerateRun`. The data are collected step by step in `B4b::SteppingAction`, and the accumulated values are entered in histograms and an ntuple event by event in `B4b::EventAction`.

**`B4b::SteppingAction`** In `UserSteppingAction()` the energy deposit and track lengths of charged particles in Absorber and Gap layers are collected and subsequently recorded in `B4b::RunData`.

**`B4b::EventAction`** In `EndOfEventAction()`, the accumulated quantities of the energy deposit and track lengths of charged particles in Absorber and Gap layers are printed and then stored in `B4b::RunData`.

**Classes in B4c (via |Geant4| sensitive detector and hits):**

**`B4c::DetectorConstruction`** In addition to materials, volumes and uniform magnetic field definitions as in `B4DetectorConstruction`, in `ConstructSDandField()` two instances of the `B4c::CalorimeterSD` class are created and associated with Absorber and Gap volumes.

**`B4c::CalorHit`** The calorimeter hit class is derived from `G4VHit`. It defines data members to store the energy deposit and track lengths of charged particles in a selected volume.

**`B4c::CalorimeterSD`** The calorimeter sensitive detector class is derived from `G4VSensitiveDetector`. Two instances of this class are created in `B4c::DetectorConstruction` and associated with Absorber and Gap volumes. In `Initialize()`, it creates one hit for each calorimeter layer and one more hit for accounting the total quantities in all layers. The values are accounted in hits in the `ProcessHits()` function, which is called by the GEANT4 kernel at each step.

**`B4c::EventAction`** In `EndOfEventAction()`, the accumulated quantities of the energy deposit and track lengths of charged particles in Absorber and Gap layers are printed and then stored in the hits collections.

**Classes in B4d (via |Geant4| scorers):**

**B4d::DetectorConstruction** In addition to materials, volumes and uniform magnetic field definitions as in `B4DetectorConstruction`, in `ConstructSDandField()` sensitive detectors of `G4MultiFunctionalDetector` type with primitive scorers are created and associated with Absorber and Gap volumes.

**B4d::EventAction** Energy deposited in crystals is summed by `G4Scorer`.
The scorers hits are saved in form of ntuples in a Root file using Geant4 analysis tools. This feature is activated in the main () function with instantiating G4TScoreNtupleWriter.
In `EndOfEventAction()`, the accumulated quantities of the energy deposit and track lengths of charged particles in Absorber and Gap layers are printed and then stored in the hits collections.

### 10.2.8 Example B5

See also Doxygen page

This example simulates a double-arm spectrometer with wire chambers, hodoscopes and calorimeters with a uniform local magnetic field.

The set of available particles and their physics processes are defined in the FTFP_BERT physics list. This GEANT4 physics list is instantiated in the main() function. It requires data files for electromagnetic and hadronic processes. See more on installation of the datasets in |Geant4| Installation Guide. The following datasets: G4LEDATA, G4LEVELGAMMADATA, G4NEUTRONXSDATA, G4SAIDXSDATA and G4ENSDFSTATEDATA are mandatory for this example.

This example also illustrates how to introduce tracking constraints like maximum step length via `G4StepLimiter`, and minimum kinetic energy, etc., via the `G4UserSpecialCuts` processes. This is accomplished by adding `G4StepLimiterPhysics` to the physics list.

**Namespace:**

Since Geant4 version 11 all example classes are defined in namespace B5.

**Classes:**

**B5::DetectorConstruction** The spectrometer consists of two detector arms. One arm provides position and timing information of the incident particle while the other collects position, timing and energy information of the particle after it has been deflected by a magnetic field centered at the spectrometer pivot point.
First arm: box filled with air, also containing:

- 1 hodoscope (15 vertical strips of plastic scintillator)
- 1 drift chamber (horizontal argon gas layers with a “virtual wire” at the center of each layer)

Second arm: box filled with air, also containing:

- 1 hodoscope (25 vertical strips of plastic scintillator)
- 1 drift chamber (5 horizontal argon gas layers with a “virtual wire” at the center of each layer)
- 1 electromagnetic calorimeter: a box sub-divided along x,y and z axes into cells of CsI (see also B5::CellParameterisation class)
- 1 hadronic calorimeter: a box sub-divided along x,y, and z axes into cells of lead, with a layer of plastic scintillator placed at the center of each cell

The magnetic field region is represented by an air-filled cylinder which contains the field (see B5::MagneticField.). The maximum step limit in the magnetic field region is also set via the G4UserLimits class in a similar way as in Example B2.
The rotation angle of the second arm and the magnetic field value can be set via the interactive command defined using the `G4GenericMessenger` class.

**B5::PrimaryGeneratorAction** The primary generator action class employs the G4ParticleGun. The primary kinematics consists of a single particle which is is sent in the direction of the first spectrometer arm.
The type of the particle and its several properties can be changed via the GEANT4 built-in commands of the `G4ParticleGun` class or this example command defined using the `G4GenericMessenger` class.

**B5::EventAction** An event consists of the generation of a single particle which is transported through the first spectrometer arm. Here, a scintillator hodoscope records the reference time of the particle before it passes through a drift chamber where the particle position is measured. Momentum analysis is performed as the particle passes through a magnetic field at the spectrometer pivot and then into the second spectrometer arm. In the second arm, the particle passes through another hodoscope and drift chamber before interacting in the electromagnetic calorimeter. Here it is likely that particles will induce electromagnetic showers. The shower energy is recorded in a three-dimensional array of CsI crystals. Secondary particles from the shower, as well as primary particles which do not interact in the CsI crystals, pass into the hadronic calorimeter. Here, the remaining energy is collected in a three-dimensional array of scintillator-lead sandwiches.

In first execution of `BeginOfEventAction()` the hits collections identifiers are saved in data members of the class and then used in `EndOfEventAction()` for accessing the hists collections and filling the accounted information in defined histograms and ntuples and printing its summary in a log file. The frequency of printing can be tuned with the built-in command `"/run/printProgress frequency".`

**`B5::RunAction`** The run action class handles the histograms and ntuples with the aid of GEANT4 analysis tools in a similar way as in Example B4. From Release 10.2 the vectors of energy deposits in Electromagnetic and Hadronic calorimeter cells are also stored in the ntuple. In difference from example B4, `G4GenericAnalysisManager` is used to demonstrate saving histograms and ntuple in two separate files.

**Hit and Sensitive Detector Classes** All the information required to simulate and analyze an event is recorded in hits. This information is recorded in the following sensitive detectors:

- Hodoscope (`B5::HodoscopeSD`, `B5::HodoscopeHit`)
  - particle time
  - strip ID, position and rotation
- Drift chamber: (`B5::DriftChamberSD`, `B5::DriftChamberHit`)
  - particle time
  - particle position
  - layer ID
- Electromagnetic calorimeter: (`B5::EmCalorimeterSD`, `B5::EmCalorimeterHit`)
  - energy deposited in cell
  - cell ID, position and rotation
- Hadronic calorimeter: (`B5::HadCalorimeterSD`, `B5::HadCalorimeterHit`)
  - energy deposited in cell
  - cell column ID and row ID, position and rotation

The hit classes include methods `GetAttDefs` and `CreateAttValues` to define and then fill extra "HepRep-style" Attributes that the visualization system can use to present extra information about the hits. For example, if you pick a `B5::HadCalorimeterHit` in OpenGL or a HepRep viewer, you will be shown the hit's "Hit Type", "Column ID", "Row ID", "Energy Deposited" and "Position".

These attributes are essentially arbitrary extra pieces of information (integers, doubles or strings) that are carried through the visualization. Each attribute is defined once in `G4AttDef` object and then is filled for each hit in a `G4AttValue` object. These attributes can also be used by commands to filter which hits are drawn: `"/vis/filtering/hits/drawByAttribute"`.

Detector Geometry and trajectories also carry HepRep-style attributes, but these are filled automatically in the base classes. HepRep is further described at: http://www.slac.stanford.edu/~perl/heprep/

## 10.3 Extended Examples

GEANT4 extended examples serve three purposes:

- testing and validation of processes and tracking,
- demonstration of GEANT4 tools, and
- extending the functionality of GEANT4.

The code for these examples is maintained as part of the categories to which they belong. Links to descriptions of the examples are listed below.

### 10.3.1 Analysis

- AnaEx01 - histogram and tuple manipulations using GEANT4 internal g4tools system
- AnaEx02 - histogram and tuple manipulations using ROOT
- AnaEx03 - usage of analysis commands for file management, writing histograms and ntuples in a file multiple times and commands for histogram deleting
- B1Con - modified basic example B1 showing how to use a Convergence Tester
- [A01] - this examples has been refactored in Example B5 in the basic set.

### 10.3.2 Biasing

- Variance Reduction - examples (*B01*, *B02* and *B03*) on variance reduction techniques and scoring and application of Reverse Monte Carlo in GEANT4 ReverseMC
- Generic biasing examples illustrate the usage of a biasing scheme implemented since version GEANT4 10.0.
  - GB01 This example illustrates how to bias process cross-sections in this scheme.
  - GB02 Illustrates a force collision scheme similar to the MCNP one.
  - GB03 Illustrates geometry based biasing.
  - GB04 Illustrates a bremsstrahlung splitting.
  - GB05 Illustrates a "splitting by cross-section" technique: a splitting-based technique using absorption cross-section to control the neutron population.
  - GB06 Illustrates the usage of parallel geometries with generic biasing.
  - GB07 Illustrates how to use the leading particle biasing option.

### 10.3.3 Common

- ReadMe - a set of classes independent on each other which can be reused in "feature" examples demonstrating just a particular feature or users applications

### 10.3.4 Electromagnetic

- TestEm0 - how to print cross-sections and stopping power used in input by the standard EM package
- TestEm1 - how to count processes, activate/inactivate them and survey the range of charged particles. How to define a maximum step size
- TestEm2 - shower development in an homogeneous material : longitudinal and lateral profiles
- TestEm3 - shower development in a sampling calorimeter : collect energy deposited, survey energy flow and print stopping power
- TestEm4 - 9 MeV point like photon source: plot spectrum of energy deposited in a single media
- TestEm5 - how to study transmission, absorption and reflection of particles through a single, thin or thick, layer.
- TestEm6 - physics list for rare, high energy, electromagnetic processes: gamma conversion and e+ annihilation into pair of muons
- TestEm7 - how to produce a Bragg curve in water phantom. How to compute dose in tallies
- TestEm8 - test of photo-absorption-ionisation model in thin absorbers, and transition radiation
- TestEm9 - shower development in a crystal calorimeter; cut-per-region
- TestEm10 - XTR transition radiation model, investigation of ionisation in thin absorbers
- TestEm11 - how to plot a depth dose profile in a rectangular box
- TestEm12 - how to plot a depth dose profile in spherical geometry : point like source
- TestEm13 - how to compute cross sections of EM processes from rate of transmission coefficient
- TestEm14 - how to compute cross sections of EM processes from direct evaluation of the mean-free path. How to plot final state
- TestEm15 - compute and plot final state of Multiple Scattering as an isolated process
- TestEm16 - simulation of synchrotron radiation

- TestEm17 - check the cross sections of high energy muon processes
- TestEm18 - energy lost by a charged particle in a single layer, due to ionization and bremsstrahlung

Table 10.4: TestEm by theme

| | |
|---|---|
| **Check basic quantities** | |
| Total cross sections, mean free paths … | Em0, Em13, Em14 |
| Stopping power, particle range … | Em0, Em1, Em5, Em11, Em12 |
| Final state : energy spectra, angular distributions | Em14 |
| Energy loss fluctuations | Em18 |
| **Multiple Coulomb scattering** | |
| as an isolated mechanism | Em15 |
| as a result of particle transport | Em5 |
| **More global verifications** | |
| Single layer: transmission, absorption, reflection | Em5 |
| Bragg curve, tallies | Em7 |
| Depth dose distribution | Em11, Em12 |
| Shower shapes, Moliere radius | Em2 |
| Sampling calorimeters, energy flow | Em3 |
| Crystal calorimeters | Em9 |
| **Other specialized programs** | |
| High energy muon physics | Em17 |
| Other rare, high energy processes | Em6 |
| Synchrotron radiation | Em16 |
| Transition radiation | Em8 |
| Photo-absorption-ionization model | Em10 |

### 10.3.5 Error Propagation

- errProp - error propagation utility

### 10.3.6 Event Generator

- exgps - illustrating the usage of the `G4GeneralParticleSource` utility
- particleGun - demonstrating three different ways of usage of `G4ParticleGun`, shooting primary particles in different cases
- userPrimaryGenerator - demonstrating how to create a primary event including several vertices and several primary particles per vertex
- HepMCEx01 - simplified collider detector using HepMC interface and stacking
- HepMCEx02 - connecting primary particles in GEANT4 with various event generators using the HepMC interface
- MCTruth - demonstrating a mechanism for Monte Carlo truth handling using HepMC as the event record
- pythia - illustrating the usage of Pythia as Monte Carlo event generator, interfaced with GEANT4, and showing how to implement an external decayer. Examples decayer6 and py8decayer

### 10.3.7 Exotic Physics

- Channeling - simulates channeling of 400 GeV/c protons in a bent crystal.
- Dmparticle - a very preliminary and simplified GEANT4 example for light dark matter (LDM) particles.
- Monopole - illustrating how to measure energy deposition in classical magnetic monopole.
- Phonon - demonstrates simulation of phonon propagation in cryogenic crystals.
- Saxs - implements the typical setup of a Small Angle X-ray Scattering (SAXS) experiment. It is meant to illustrate the usage of molecular interference (MI) of Rayleigh (coherent) scattering of photons inside the matter.
- UCN - simulates the passage of ultra-cold neutrons (UCN) in a hollow pipe.

### 10.3.8 Fields

- BlineTracer - tracing and visualizing magnetic field lines
- field01 - tracking using magnetic field and field-dependent processes
- field02 - tracking using electric field and field-dependent processes
- field03 - tracking in a magnetic field where field associated with selected logical volumes varies
- field04 - definition of overlapping fields either magnetic, electric or both
- field05 - demonstration of "spin-frozen" condition, how to cancel the muon g-2 precession by applying an electric field
- field06 - exercising the capability of tracking massive particles in a gravity field

### 10.3.9 Geant3 to Geant4

- General ReadMe - converting simple geometries in Geant3.21 to their GEANT4 equivalents (example *clGeometry*)

### 10.3.10 Geometry

- General ReadMe
- transforms - demonstrating various ways of definition of 3D transformations for placing volumes
- vecGeomNavigation - demonstrating integration of the navigation elements of VecGeom

### 10.3.11 Hadronic

- Hadr00 - example demonstrating the usage of G4PhysListFactory to build physics lists and usage of G4HadronicProcessStore to access the cross sections
- Hadr01 - example based on the application IION developed for simulation of proton or ion beam interaction with a water target. Different aspects of beam target interaction are included
- Hadr02 - example application providing simulation of ion beam interaction with different targets. Hadronic aspects of beam target interaction are demonstrated including longitudinal profile of energy deposition, spectra of secondary particles, isotope production spectra.
- Hadr03 - example demonstrating how to compute total cross section from the direct evaluation of the mean free path, how to identify nuclear reactions and how to plot energy spectrum of secondary particles
- Hadr04 - example focused on neutronHP physics, especially neutron transport, including thermal scattering
- Hadr05 - examples of hadronic calorimeters
- Hadr06 - demonstrates survey of energy deposition and particle's flux from a hadronic cascade
- Hadr07 - demonstrates survey of energy deposition and particle's flux from a hadronic cascade. Show how to plot a depth dose profile in a rectangular box.
- Hadr08 - demonstrates how to use "generic biasing" to get the following functionality which is currently not available directly in the Geant4 hadronic framework. We want to use the physics list FTFP_BERT everywhere in

our detector, except that in one (or more) logical volume(s) we want to use a different combination of hadronic models, e.g. FTFP + INCLXX (instead of the default FTFP + BERT), for the final-state generation.
- Hadr09 - demonstrates how to use Geant4 as a generator for simulating inelastic hadron-nuclear interactions. Notice that the Geant4 run-manager is not used. See the README file (provided with the example) for more information.
- Hadr10 - aims to test the treatment of decays in Geant4. In particular, we want to test the decays of the tau lepton, charmed and bottom hadrons, and the use of pre-assigned decays.
- FissionFragment - This example demonstrates the Fission Fragment model as used within the neutron_hp model. It will demonstrate the capability for fission product containment by the cladding in a water moderated subcritical assembly. It could also be further extended to calculate the effective multiplication factor of the subcritical assembly for various loading schemes.
- NeutronSource - NeutronSource is an example of neutrons production. It illustrates the cooperative work of nuclear reactions and radioactive decay processes. It surveys energy deposition and particle's flux. It uses PhysicsConstructor objects.
- ParticleFluence - New set of extended examples, implementing different setups showing how to score particle fluences.

## 10.3.12 Medical Applications

- DICOM - geometry set-up using the GEANT4 interface to the DICOM image format
- DICOM2 - inheritance from the DICOM example, method for memory savings, scoring into a sequential container instead of an associative container, accumulating the scoring with a statistics class instead of a simple floating point, and generic iteration over the variety of scoring container storage variants
- electronScattering - benchmark on electron scattering
- electronScattering2 - benchmark on electron scattering (second way to implement the same benchmark as the above)
- fanoCavity - dose deposition in an ionization chamber by a monoenergetic photon beam
- fanoCavity2 - dose deposition in an ionization chamber by an extended one-dimensional monoenergetic electron source
- GammaTherapy - gamma radiation field formation in water phantom by electron beam hitting different targets
- radiobiology - an application realized for dosimetric and radiobiological applications of proton and ion beams
- dna - Set of examples using the GEANT4-DNA physics processes and models.
  - AuNP - Simulation of the track structure of electrons in a microscopic gold volume.
  - UHDR - Shows how to activate the mesoscopic model in chemistry and combine with SBS mode. It allows to simulate chemical reactions long time (beyond 1 us) of post- irradiation.
  - chem1 - Simple activation of the chemistry module.
  - chem2 - Usage of TimeStepAction in the chemistry module.
  - chem3 - Activate the full interactivity with the chemistry module.
  - chem4 - Simulation of G radiochemical yields with the chemistry module.
  - chem5 - A variation of the chem4 example, using preliminary G4EmDNAPhysics_option8 and G4EmDNAChemistry_option1 constructors.
  - chem6 - Scoring of the radiochemical yield G as a function of time and LET.
  - clustering - Clustering application for direct damage extraction.
  - dnadamage1 - Simulation of damage on a chromatin fiber.
  - dnadamage2 - Simulation of scoring of plasmid DNA strand breaks using the IRT method. It extends the chem6 example by adding DNA molecule information and the scoring of Strand Breaks.
  - dnaphysics - Simulation of track structures in liquid water using the GEANT4-DNA physics processes and models.
  - icsd - Use of cross section models for DNA materials.
  - jetcounter - Simulation of a typical experiment with the Jet Counter nanodosemeter
  - mfp - Simulation of mean free path in liquid water.
  - microdosimetry - Simulation of the track of a 5 MeV proton in liquid water. GEANT4 standard EM models are used in the World volume while GEANT4-DNA models are used in a Target volume, declared as a

Region.
  – microprox - Computation of proximity functions in liquid water.
  – microyz - Simulation of microdosimetry spectra.
  – moleculardna - Simulation of physics, physico-chemistry and chemistry processes in DNA geometries. See more details in *the molecularDNA docs <https://geant4-dna.github.io/molecular-docs/>*
  – neuron - Irradiation of a realistic neuron cell.
  – pdb4dna - Usage of the Protein Data Bank (PDB) file format to build geometries.
  – range - Simulation of ranges.
  – scavenger - Simulation of the scavenging process in chemistry using the deterministic treatment of the IRT model.
  – slowing - Simulation of slowing down spectra.
  – scavenger - Simulation of the scavenging process in chemistry using the deterministic treatment of the IRT model.
  – splitting - Acceleration of Geant4-DNA physics simulations by particle splitting.
  – spower - Simulation of stopping power.
  – svalue - Simulation of S-values in spheres of liquid water using the GEANT4-DNA physics processes and models.
  – wholeNuclearDNA - Description of the full nucleus of a biological cell.
  – wvalue - Simulation of W-values in liquid water using the GEANT4-DNA physics processes and models.

## 10.3.13 Optical Photons

- General ReadMe
- OpNovice - simulation of optical photons generation and transport. (It was moved in extended examples from novice/N06 with removal of novice examples.)
- OpNovice2 - investigate optical properties and parameters; details of optical photon boundary interactions on a surface, optical photon generation and transport
- LXe - optical photons in a liquid xenon scintillator
- WLS - application simulating the propagation of photons inside a Wave Length Shifting (WLS) fiber

## 10.3.14 Parallel Computing

- General ReadMe
- MPI - interface and examples of applications (*exMPI01*, *exMPI02*, exMPI03* and *exMPI04*) parallelized with different MPI compliant libraries, such as LAM/MPI, MPICH2, OpenMPI, etc.
- TBB - demonstrate how to interface a simple application with the Intel Threading Building Blocks library (TBB), and organise MT event-level parallelism as TBB tasks
- ThreadsafeScorers - demonstrates a very simple application where an energy deposit and # of steps is accounted in thread-local (i.e. one instance per thread) hits maps with underlying types of plain-old data (POD) and global (i.e. one instance) hits maps with underlying types of atomics.
- TopC - set of examples (*ParN02* and *ParN04*) derived from `novice` using parallelism at event level with the TopC application

### 10.3.15 Parameterisations

- Par01 - Demonstrates the use of parameterisation facilities. (It was moved in extended examples from novice/N05 with removal of novice examples.)
- Par02 - Shows how to do "track and energy smearing" in GEANT4, in order to have a very fast simulation based on assumed detector resolutions.
- Par03 - Demonstrates how to create multiple energy deposits using helper class `G4FastSimHitMaker`. Thanks to storing hits in the same hit collection (using one sensitive detector class) whether they originated in the full or fast simulation, this example allows to perform the same analysis on both outputs and compare the results.
- *Par04* (doxygen) - Demonstrates how to use machine learning techniques for the fast simulation of calorimeters, and how to incorporate inference libraries into C++ framework.
- Gflash - Examples set (*gflash1*, *2*, *3*, *gflasha*) demonstrating the use of the GFLASH parameterisation library. It uses the GFLASH equations (hep-ex/0001020, Grindhammer & Peters) to parametrise electromagnetic showers in matter.

#### Par04

Par04 example focuses on application of Machine Learning (ML) techniques to fast simulation of calorimeters. Its main goal is to demonstrate how to do ML inference using Geant4.

In order to show how to perform ML inference, Par04 contains an ML model. It was trained externally (with Python) on dataset from standard (full) simulation done with this example. It is not optimized as its accuracy is not the concern of this example.

#### Installation

Par04 example depends on external libraries used for the ML inference. Currently, following libraries can be used:

- ONNX Runtime,
- LWTNN.

If those libraries are not available, Par04 example can still be built to run standard (full) simulation. Such application can be used for instance to simulate the dataset used for ML training. The search for external libraries is by default switched on, but it can be switched off by setting the CMake variable `INFERENCE_LIB` to OFF.

To build the application you need to:

```
$ cmake <Par04_SOURCE>
$ make
```

Replace `<Par04_SOURCE>` with the path to Par04 example source directory. CMake will look for inference libraries unless you use `-DINFERENCE_LIB=OFF`. You can point manually to them using `CMAKE_PREFIX_PATH` variable.

For instance, to run the example on `lxplus8` you could use:

```
$ source /cvmfs/sft.cern.ch/lcg/contrib/gcc/11.1.0/x86_64-centos8-gcc11-opt/setup.sh
$ cmake <Par04_SOURCE> -DCMAKE_PREFIX_PATH="/cvmfs/sft.cern.ch/lcg/releases/onnxruntime/1.8.0-
↪47224/x86_64-centos8-gcc11-opt;/cvmfs/sft.cern.ch/lcg/releases/lwtnn/2.11.1-72aca/x86_64-
↪centos8-gcc11-opt/"
```

### How to run

Example can be run in batch mode, as well as in the interactive mode with the hit visualization.

To run the example in the batch mode you need to specify the macro (options) passing it with `-m`:

```
$ ./examplePar04 -m <MACRO_FILE>
```

To run the example in the interactive mode:

```
$ ./examplePar04
```

Example contains following macro files:

- `examplePar04.in` - runs full simulation.
- `examplePar04_onnx.in` - runs inference with ONNX Runtime. It is installed in the build directory only if ONNX Runtime is found by CMake.
- `examplePar04_lwtnn.in` - runs inference with LWTNN. It is installed in the build directory only if LWTNN is found by CMake.
- `vis.mac` - this macro is used in the interactive mode.

### ML fast calorimeter simulation

### Calorimeter

Calorimeter used in this example is a simple setup of concentric cylinders of active and passive material. It can be configured from the macro using the UI commands.

Energy deposits are scored in the detector using the cylindrical readout structure, centered around the particle momentum, as shown in Fig. 10.1. Dimensions of the readout structure (number and size of cells) can be also configured using the UI commands.

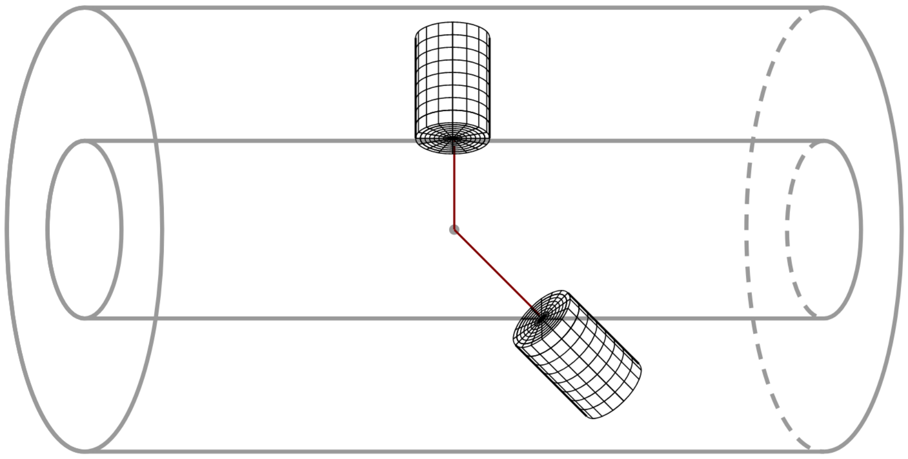


Fig. 10.1: Energy deposits are scored in cylindrical readout around particle momentum.

This example uses silicon as an active material, and tungsten as the passive absorber. 90 layers of both materials are placed, with 1.4 mm of tungsten and 0.3 mm of silicon. Size of the cylindrical readout has been optimized to contain (on average) 95 % of energy of 1 TeV electrons. The size of single cell has been chosen to correspond to (approximately) 0.25 Moliere radius and 0.5 radiation length.

Detector and readout (mesh) used in the example macros are configured with following commands:

```
# Detector Construction
/Par04/detector/setDetectorInnerRadius 80 cm
/Par04/detector/setDetectorLength 2 m
/Par04/detector/setNbOfLayers 90
/Par04/detector/setAbsorber 0 G4_W 1.4 mm false
/Par04/detector/setAbsorber 1 G4_Si 0.3 mm true
## 2.325 mm of tungsten =~ 0.25 * 9.327 mm = 0.25 * R_Moliere
/Par04/mesh/setSizeOfRhoCells 2.325 mm
## 2 * 1.4 mm of tungsten =~ 0.65 X_0
/Par04/mesh/setSizeOfZCells 3.4 mm
/Par04/mesh/setNbOfRhoCells 18
/Par04/mesh/setNbOfPhiCells 50
/Par04/mesh/setNbOfZCells 45
```

Particle momentum that is used to define the orientation and the placement of the cylindrical readout (which will differ from particle to particle) is measured at the entrance to the calorimeter. This measurement can be done in several different ways, and this example is using a fast simulation model (`Par04DefineMeshModel`) that is triggered at the entrance to the calorimeter and that sets up particle entrance position and momentum in the event information `Par04EventInformation` (since single particle events are used). For simplicity of the example `Par04DefineMeshModel` is attached to the same region as `Par04InferenceModel`. It has two direct consequences:

- In fast simulation order of activation of those model must be respected (first the model which sets up the readout properties, and then the model which kills a particle and creates deposits).
- Full simulation is slowed down by the large region of calorimeter and presence of fast simulation model that is called (to check if particle enters the calorimeter) for every electron or photon. For realistic use case this model should be attached to very thin cylindrical region located just *before* the entrance to the calorimeter.

### Output data

Cylindrical readout is used in the sensitive detector `Par04SensitiveDetector` which accumulates energy from the event in the collection of hits, ignoring the deposits outside of the cylindrical mesh. At the end of each event, hit collection is saved to ntuple and stored in ROOT files. Additionally, simple analysis of the shower shape is performed and histograms are saved alongside the ntuple (technically, if G4 is run in sequential mode, otherwise multiple ROOT files are produced because histograms are merged while ntuples are not).

Structure of the output file(s) is the following:

```
10GeV_100events_fullsim.root
├── events
│   ├── (D) EnergyMC
│   ├── (VD) EnergyCell
│   ├── (VI) rhoCell
│   ├── (VI) phiCell
│   ├── (VI) zCell
│   └── (D) SimTime
├── (H) energyParticle
├── (H) energyDeposited
├── (H) energyRatio
├── (H) time
├── (H) longProfile
├── (H) transProfile
├── (H) longFirstMoment
├── (H) transFirstMoment
├── (H) longSecondMoment
├── (H) transSecondMoment
└── (H) hitType
```

Where:

- (D) is a double value,
- (VD) is a vector of double values,
- (VI) is a vector of integers,
- (H) is a histogram.

Ntuple `events` contains information on energy of primary particle (in units of MeV), and vector of energy deposits: cylindrical coordinates and energy in default GEANT4 units (rad for phi, mm for rho and z, MeV for energy). Only deposits above hard-coded threshold (E>0.5 keV) are stored in the file. Histograms are created in a simple post-event analysis. Please note that for readability, energy of primary particle and total deposited energy are plotted in units of GeV. Moreover, all hits with non-zero energy are taken into account (there is no minimal energy threshold). Hit type can be used to distinguish between full and fast simulation. Hit with ID=0 is used for full simulation, and ID=1 for fast simulation.

## ML model

The model used in this example was trained externally (in Python) on data from this examples' full simulation. It is a Variational Autoencoder (VAE): a deep learning generative model. The VAE is composed of two stacked deep neural networks acting as encoder and decoder. The encoder learns a mapping from the input space to an unobserved or latent space in which a lower dimensional representation of the full simulation is learned. The decoder learns the inverse mapping, thus reconstructing the original input from this latent representation. The encoded distributions are constrained to be Gaussian distributions and the encoder is tasked to return the mean and the covariance matrix that describe these distributions.

The loss function that is optimized during the training of the VAE is composed of a regularisation loss to minimize Kulback-Leibler divergence between encoded distributions and prior Gaussian distributions, a reconstruction loss to minimize the error by computing the binary cross-entropy between the input and its reconstruction version using the latent representation.

The VAE architecture used in this example comprises 4 hidden layers with width of 100,50,20,14 and 14,20,50,100 for the encoder and decoder respectively as shown in the figure below.

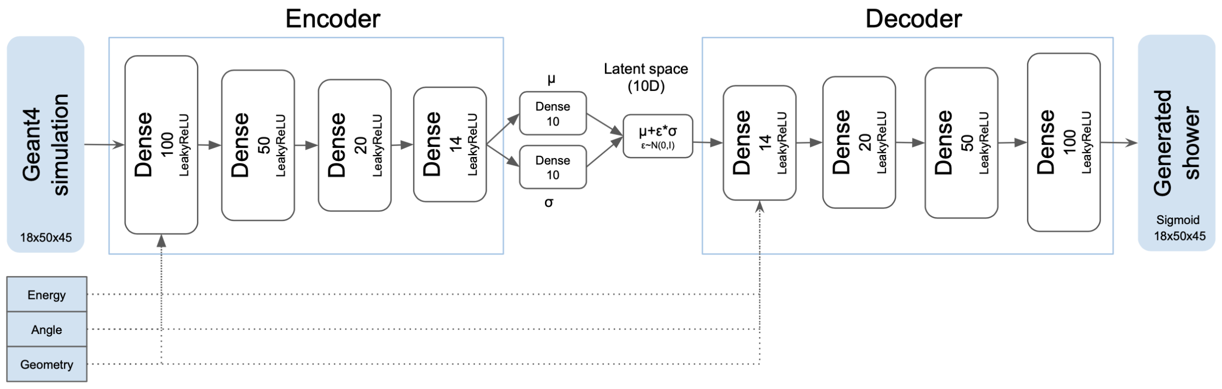


Fig. 10.2: VAE model architecture.

The full simulation samples for the two detector geometries (Si/W as used in the example, and additionally 1.2 mm Scintillator/4.4 mm Pb) are showers of electron particles generated with an energy range from 1 GeV to 1 TeV (in powers of 2) and angles from 50 to 90 degrees (in a step of 10 degrees). 90 degrees means perpendicular to the z-axis. The VAE is conditioned on the three parameters.

The three figures below show the validation plots comparing the full simulation to the (ML) fast simulation after using the inference with ONNX. The plots show the longitudinal (Fig. 10.3), transverse profiles (Fig. 10.4) and simulation time (Fig. 10.5) for 64 GeV particles with an angle of 90 degrees.

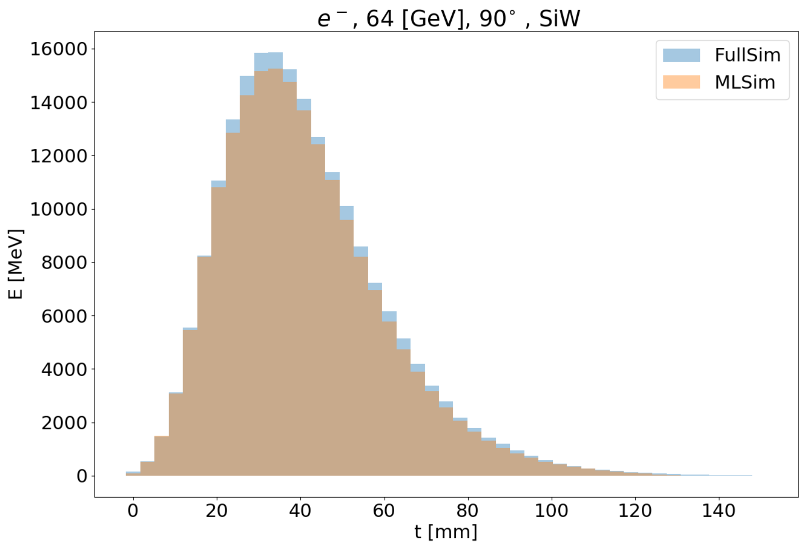


Fig. 10.3: Longitudinal profile for 64 GeV electrons.

### Inference

In this example, the input inference vector is constructed by sampling from a 10D Gaussian distribution. The condition vector comprises condition values of energy and angle of a particle and two values encoding the calorimeter geometry. The condition value of the energy of the particle is normalized to the maximum energy point in the range of training. For the angle, the value in degrees is also normalized to the maximum angle point in the range of training. The model was trained on two detector geometries and the conditioning of the geometry used is a one hot encoding vector with [0,1] for SiW geometry and [1,0] for SciPb geometry. After running the inference, the values are rescaled back by the energy of the particle.

### How to run inference of user models

This example was designed to facilitate integration of user model in Geant4 toolkit. Figure Fig. 10.6 shows how fast simulation components were implemented. Fast simulation model `Par04InferenceModel` is responsible for taking the particle out of full simulation, and creation of energy deposits with energy and position returned by `Par04InferenceSetup`. This is the model(user) specific class that should configure the ML model, run the inference using one of the inference libraries (configured in the example with UI commands thanks to `Par04InferenceSetupMessenger`), do the energy postprocessing, and finally be aware where to place energy inside of the detector. Interface to inference libraries (`Par04InferenceInterface`) is designed in a model-agnostic way. The inference works for any trained model by sampling N points from a predefined distribution where N represents the size of the input inference vector.

For a specific application, the user should therefore only change `Par04InferenceSetup` class, where all inference parameters are defined. These parameters include the name of the inference library, the path and name of the inference model, the size of the input inference vector (latent vector size and condition vector size). In this class the user can

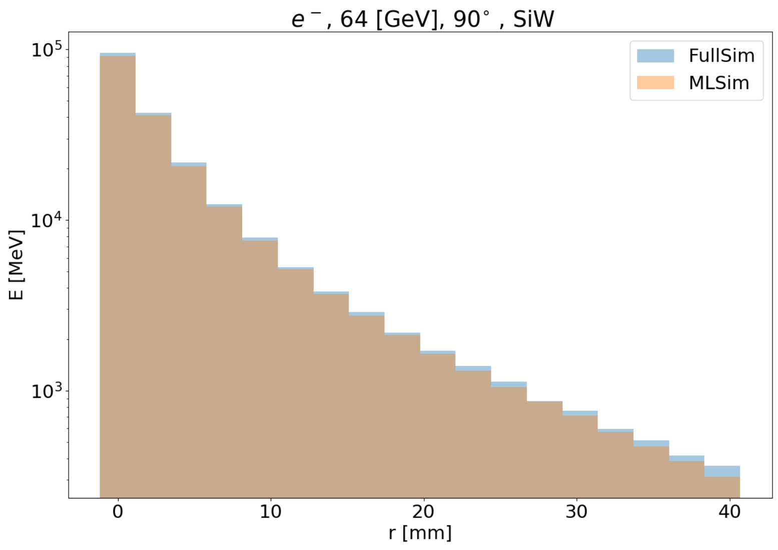


Fig. 10.4: Transvers profile for 64 GeV electrons.

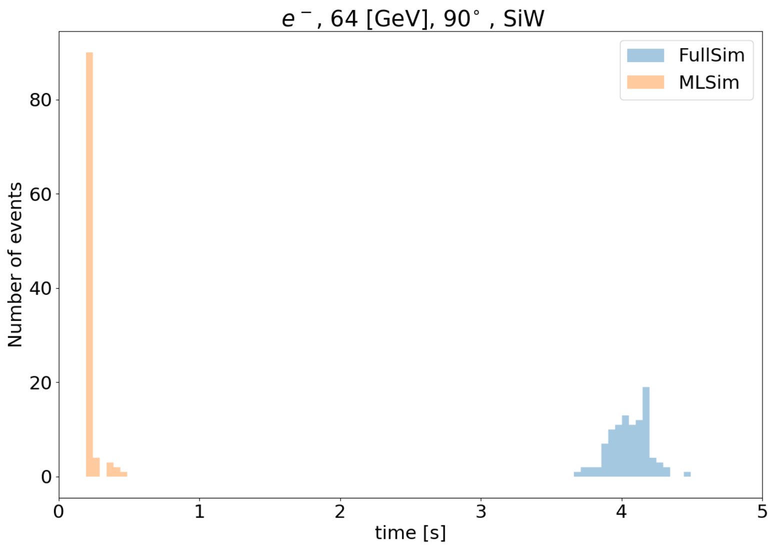


Fig. 10.5: Simulation time for 64 GeV electrons.

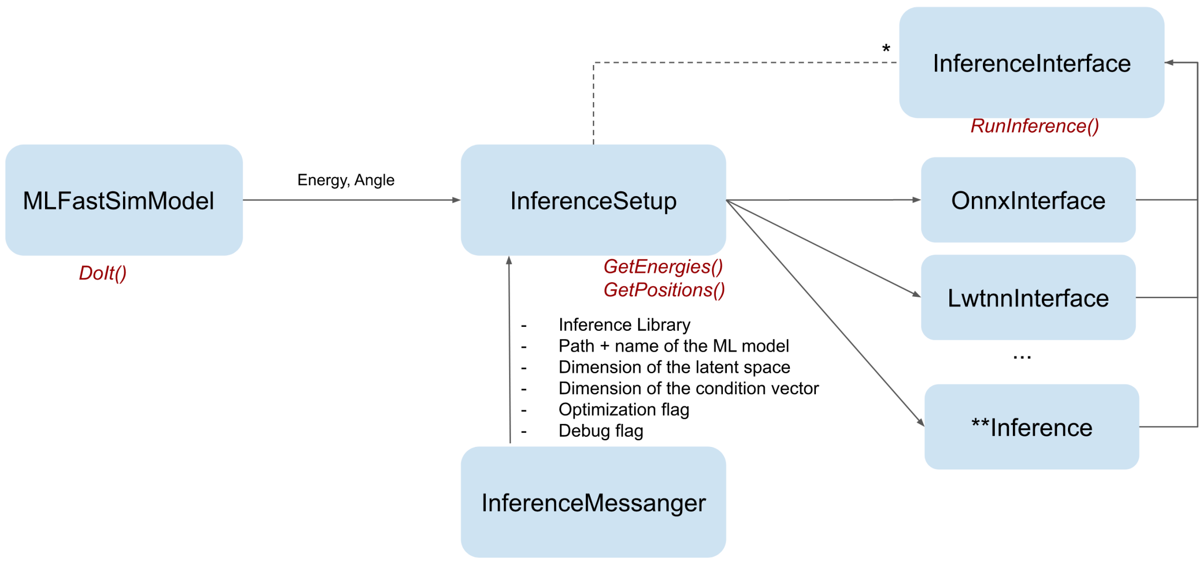


Fig. 10.6: Diagram of the classes used for fast simulation with ML inference.

define the input inference vector including the vector of conditions, run the inference and also apply a post processing step to retrieve the original energy range if the model was trained on any specific preprocessed values. The important part is also assignment of positions to the energy deposits. Fast simulation model `Par04InferenceModel` will use those positions to place the deposits, therefore it needs to be a position of the sensitive detector. For this reason, in Par04 example, where simple center of cell positions are used, both materials (active and passive) is considered as sensitive for the fast simulation runs.

### Inference with LWTNN

Lightweight Trained Neural Network (LWTNN) supports scikit-learn and Keras models where this model is saved as two separate files of the architecture (in JSON) and the weights (in HDF5). The trained model can be saved into these two separate files with:

```python
# save the architecture in a JSON file
with open('architecture.json', 'w') as arch_file:
    arch_file.write(model.to_json())
# save the weights as an HDF5 file
model.save_weights('weights.h5')
```

After building the LWTNN code available at this link, run the `kerasfunc2json` python script (available in `lwtnn/converters/`) to generate a template file of your functional model input variables by calling:

```
$ kerasfunc2json.py architecture.json weights.h5 > inputs.json
```

And run again `kerasfunc2json` script to get your output file that would be used for the inference in C++:

```
$ kerasfunc2json.py architecture.json weights.h5 inputs.json > Generator.json
```

`Par04LwtnnInference` is the class called if the user chooses the LWTNN library. The object that will do the computation in this class is a `LightweightGraph`, initialized from `Generator.json` file. The inference is based on evaluating the graph using the input inference vector constructed in `Par04InferenceSetup`.

#### Inference with ONNX runtime

Open Neural Network Exchange (ONNX) runtime supports models from Tensorflow/Keras, PyTorch, TFLite, scikit-learn and other frameworks. For a Keras model for example, to save it into an ONNX, you can first save it as HDF5 file with:

```
model.save("model.h5")
```

This model is then converted into an ONNX format using keras2onnx with:

```
# Create the Keras model
kerasModel = tensorflow.keras.models.load_model("model.h5")
# Convert Keras model into an ONNX model
onnxModel = keras2onnx.convert_keras( kerasModel , 'name' )
# Save the ONNX model
keras2onnx.save_model(onnxModel, 'Generator.onnx')
```

`Par04OnnxInference` is the class that is called if the user chooses ONNX. It creates an environment which manages an internal thread pool and creates as well the inference session for the model. This session runs the inference using the input vector constructed in `Par04InferenceSetup`.

### 10.3.16 Persistency

- General ReadMe
- GDML - examples set (*G01*, *G02*, *G03* and *G04*) illustrating import and export of a detector geometry with GDML, and how to extend the GDML schema or use the auxiliary information field for defining additional persistent properties
- P01 - storing calorimeter hits using reflection mechanism with Root
- P02 - storing detector description using reflection mechanism with Root
- P03 - illustrating import and export of a detector geometry using ASCII text description and syntax

### 10.3.17 Physics lists

- General ReadMe
- factory - demonstrates the usage of G4PhysListFactory to build the concrete physics list.
- extensibleFactory - demonstrates the usage of extensible g4alt::G4PhysListFactory to build a concrete physics list. It also demonstrates the setting of an alternative "default" physics list; extending existing lists by adding/replacing physics constructors; and extending the factory with user supplied physics lists.
- genericPL - demonstrates the usage of G4GenericPhysicsList to build the concrete physics list at the run time.

### 10.3.18 Polarisation

- Pol01 - interaction of polarized beam (e.g. circularly polarized photons) with polarized target

### 10.3.19 Radioactive Decay

- rdecay01 - demonstrating basic functionality of the `G4RadioactiveDecay` process
- rdecay02 (Exrdm) - decays of radioactive isotopes as well as induced radioactivity resulted from nuclear interactions
- Activation - compute and plot time evolution of each nuclide in an hadronic cascade. Compute and plot activity of emerging particles.

### 10.3.20 Run & Event

- RE01 - information between primary particles and hits and usage of user-information classes
- RE02 - simplified fixed target application for demonstration of primitive scorers
- RE03 - use of UI-command based scoring; showing how to create parallel world(s) for defining scoring mesh(es)
- RE04 - demonstrating how to define a layered mass geometry in parallel world
- RE05 - demonstrating interfacing to the PYTHIA primary generator, definition of a 'readout' geometry, event filtering using the stacking mechanism. (It was moved in extended examples from novice/N04 with removal of novice examples.)
- RE06 - demonstrating how to modify part of the geometry setup at run-time, detector description parameterisation by materials, sharing of a sensitive detector definition for different sub-detectors, different geometrical regions definition with different production thresholds, customization of the G4Run (It was moved in extended examples from novice/N07 with removal of novice examples.)
- RE07 - based on extended/electromagnetic/TestEm3, this example demonstrates how to register specialized tracking managers for a particle or a set of particles.

### 10.3.21 Visualization

- General ReadMe - examples (*perspective*, *standalone* and *userVisAction*) of customisation for visualization
- movies - illustrating how to create a movie

## 10.4 Advanced Examples

GEANT4 advanced examples illustrate realistic applications of GEANT4 in typical experimental environments. Most of them also show the usage of analysis tools (such as histograms, ntuples and plotting), various visualization features and advanced user interface facilities, together with the simulation core.

**Note:** Maintenance and updates of the code is under the responsibility of the authors. No guarantee can be provided on the functionality and the accuracy deriving from the simulation results.

The advanced examples are fully documented here, below is a summary of what is available:

Table 10.5: Advanced Examples

| Example Name | Short Description |
|---|---|
| air_shower | Modelling of the ULTRA experiment, EUSO mission |
| ams_Ecal | Modelling of the electromagnetic Calorimeter (ECAL) of the AMS-02 experiment |
| brachytherapy | Calculation of dose in a phantom, in the context of brachytherapy |

continues on next page

Table 10.5 – continued from previous page

| Example Name | Short Description |
|---|---|
| CaTS | Demonstration of the *G4Opticks* hybrid workflow for the creation and propagation of optical photons on GPUs |
| ChargeExchangeMC | Simulation of hadronic physics experiments of the Petersburg Nuclear Physics Institute (PNPI, Russia) |
| composite_calorimeter | Example of a test-beam simulation used by the CMS |
| doiPET | Modelling of a PET scintillator system |
| eFLASH_radiotherapy | Modelling of a FLASH radiotherapy beamline |
| eRosita | Modelling of eROSITA astronomical X-ray full-sky survey mission on-board the Spectrum-X-Gamma space mission |
| exp_microdosimetry | Modelling of detectors and their response for microdosimetry for radiation protection in space |
| fastAerosol | Development of a custom geometry class for accurately and efficiently simulating aerosols with many droplets |
| gammaknife | Simulation of an advanced device for Stereotactic Radiosurgery |
| gammaray_telescope | Model of a typical telescope for gamma ray analysis in the context of space exploration |
| gorad | Turn-key application for radiation analysis and spacecraft design built on top of Geant4 |
| hadrontherapy | Model of hadrontherapy beamlines |
| HGCal_testbeam | Demonstration of a high-end High Energy Physics test beam setup, for the endcap electromagnetic calorimeter of the CMS detector CERN-LHCC-2017-023 |
| human_phantom | Calculation of dose in analytical anthropomorphic phantoms |
| ICRP110_HumanPhantoms | Calculation of dose in ICRP110 anthorpomorphic phantoms |
| ICRP145_HumanPhantoms | Calculation of dose in ICRP145 anthorpomorphic phantoms |
| iort_therapy | Model of a typical Intraoperative Radiation Therapy beamline |
| lAr_calorimeter | Simulation of the Forward Liquid Argon Calorimeter (FCAL) of the ATLAS Detector, CERN, Switzerland |
| medical_linac | Model of a typical medical linear accelerator for Intensity Modulated Radiation Therapy (IMRT) |
| microbeam | Simulation of the microbeam cellular irradiation beam line installed on the AIFIRA electrostatic accelerator facility located at LP2i Bordeaux, France |
| microelectronics | Demonstration on how to activate track structure physics models for electrons in a silicon microelectronics device |
| nanobeam | Simulation of the beam optics of the "nanobeam line" installed on the AIFIRA electrostatic accelerator facility located at LP2i Bordeaux, France |
| purging_magnet | Modelling of electrons traveling through a 3D magnetic field in the radiotherapy context |

continues on next page

Table 10.5 – continued from previous page

| Example Name | Short Description |
|---|---|
| STCyclotron | Model of the solid target of the South Australian Health and Medical Research Institute (SAHMRI), Adelaide, South Australia |
| stim_pixe_tomography | Simulation of three dimensional proton micro-tomography |
| underground_physics | Example of an underground dark matter experiment. More details are provided in the README file accompanying the example |
| xray_fluorescence | Example reproducing various setups for PIXE and XRF experiments. More details are provided in the README file accompanying the example |
| xray_telescope | Simulation of a typical X-ray telescope for space exploration |
| xray_TESdetector | Application of Geant4 in a space environment. Model of an X-ray detector derived from the X-IFU, the X-ray spectrometer designed and developed by the European Space Agency (ESA) for use on the ATHENA telescope. |
| Xray_SiliconPoreOptics | Model of a single reflective pore used to simulate on a smaller scale the effect of the millions of pores forming the mirror of the ATHENA Silicon Pore Optics (SPO). |

## 10.5 Novice Examples

The old “novice” set of examples is now replaced with a new “basic” set, covering the most typical use-cases of a GEANT4 application with keeping simplicity and ease of use.

The source code of the last version of the novice examples set (in 9.6.p02 release) can be viewed in the GEANT4 LXR code browser.

The new location of each example in 10.0 release:

- N01 - removed
- N02 - basic/B2
- N03 - basic/B4
- N04 - extended/runAndEvent/RE05
- N05 - extended/parameterisations/Par01
- N06 - extended/optical/OpNovice
- N07 - extended/runAndEvent/RE06

CHAPTER

# ELEVEN

# APPENDIX

## 11.1 GEANT4 Material Database

### 11.1.1 Simple Materials (Elements)

| Z | Name | density(g/cm^3) | I(eV) |
|---|---|---|---|
| 1 | G4_H | 8.3748e-05 | 19.2 |
| 2 | G4_He | 0.000166322 | 41.8 |
| 3 | G4_Li | 0.534 | 40 |
| 4 | G4_Be | 1.848 | 63.7 |
| 5 | G4_B | 2.37 | 76 |
| 6 | G4_C | 2 | 81 |
| 7 | G4_N | 0.0011652 | 82 |
| 8 | G4_O | 0.00133151 | 95 |
| 9 | G4_F | 0.00158029 | 115 |
| 10 | G4_Ne | 0.000838505 | 137 |
| 11 | G4_Na | 0.971 | 149 |
| 12 | G4_Mg | 1.74 | 156 |
| 13 | G4_Al | 2.699 | 166 |
| 14 | G4_Si | 2.33 | 173 |
| 15 | G4_P | 2.2 | 173 |
| 16 | G4_S | 2 | 180 |
| 17 | G4_Cl | 0.00299473 | 174 |
| 18 | G4_Ar | 0.00166201 | 188 |
| 19 | G4_K | 0.862 | 190 |
| 20 | G4_Ca | 1.55 | 191 |
| 21 | G4_Sc | 2.989 | 216 |
| 22 | G4_Ti | 4.54 | 233 |
| 23 | G4_V | 6.11 | 245 |
| 24 | G4_Cr | 7.18 | 257 |
| 25 | G4_Mn | 7.44 | 272 |
| 26 | G4_Fe | 7.874 | 286 |
| 27 | G4_Co | 8.9 | 297 |
| 28 | G4_Ni | 8.902 | 311 |
| 29 | G4_Cu | 8.96 | 322 |
| 30 | G4_Zn | 7.133 | 330 |
| 31 | G4_Ga | 5.904 | 334 |
| 32 | G4_Ge | 5.323 | 350 |

continues on next page

Table 11.1 – continued from previous page

| Z | Name | density(g/cm^3) | I(eV) |
|---|---|---|---|
| 33 | G4_As | 5.73 | 347 |
| 34 | G4_Se | 4.5 | 348 |
| 35 | G4_Br | 0.0070721 | 343 |
| 36 | G4_Kr | 0.00347832 | 352 |
| 37 | G4_Rb | 1.532 | 363 |
| 38 | G4_Sr | 2.54 | 366 |
| 39 | G4_Y | 4.469 | 379 |
| 40 | G4_Zr | 6.506 | 393 |
| 41 | G4_Nb | 8.57 | 417 |
| 42 | G4_Mo | 10.22 | 424 |
| 43 | G4_Tc | 11.5 | 428 |
| 44 | G4_Ru | 12.41 | 441 |
| 45 | G4_Rh | 12.41 | 449 |
| 46 | G4_Pd | 12.02 | 470 |
| 47 | G4_Ag | 10.5 | 470 |
| 48 | G4_Cd | 8.65 | 469 |
| 49 | G4_In | 7.31 | 488 |
| 50 | G4_Sn | 7.31 | 488 |
| 51 | G4_Sb | 6.691 | 487 |
| 52 | G4_Te | 6.24 | 485 |
| 53 | G4_I | 4.93 | 491 |
| 54 | G4_Xe | 0.00548536 | 482 |
| 55 | G4_Cs | 1.873 | 488 |
| 56 | G4_Ba | 3.5 | 491 |
| 57 | G4_La | 6.154 | 501 |
| 58 | G4_Ce | 6.657 | 523 |
| 59 | G4_Pr | 6.71 | 535 |
| 60 | G4_Nd | 6.9 | 546 |
| 61 | G4_Pm | 7.22 | 560 |
| 62 | G4_Sm | 7.46 | 574 |
| 63 | G4_Eu | 5.243 | 580 |
| 64 | G4_Gd | 7.9004 | 591 |
| 65 | G4_Tb | 8.229 | 614 |
| 66 | G4_Dy | 8.55 | 628 |
| 67 | G4_Ho | 8.795 | 650 |
| 68 | G4_Er | 9.066 | 658 |
| 69 | G4_Tm | 9.321 | 674 |
| 70 | G4_Yb | 6.73 | 684 |
| 71 | G4_Lu | 9.84 | 694 |
| 72 | G4_Hf | 13.31 | 705 |
| 73 | G4_Ta | 16.654 | 718 |
| 74 | G4_W | 19.3 | 727 |
| 75 | G4_Re | 21.02 | 736 |
| 76 | G4_Os | 22.57 | 746 |
| 77 | G4_Ir | 22.42 | 757 |
| 78 | G4_Pt | 21.45 | 790 |
| 79 | G4_Au | 19.32 | 790 |
| 80 | G4_Hg | 13.546 | 800 |
| 81 | G4_Tl | 11.72 | 810 |

continues on next page

Table 11.1 – continued from previous page

| Z | Name | density(g/cm^3) | I(eV) |
|---|---|---|---|
| 82 | G4_Pb | 11.35 | 823 |
| 83 | G4_Bi | 9.747 | 823 |
| 84 | G4_Po | 9.32 | 830 |
| 85 | G4_At | 9.32 | 825 |
| 86 | G4_Rn | 0.00900662 | 794 |
| 87 | G4_Fr | 1 | 827 |
| 88 | G4_Ra | 5 | 826 |
| 89 | G4_Ac | 10.07 | 841 |
| 90 | G4_Th | 11.72 | 847 |
| 91 | G4_Pa | 15.37 | 878 |
| 92 | G4_U | 18.95 | 890 |
| 93 | G4_Np | 20.25 | 902 |
| 94 | G4_Pu | 19.84 | 921 |
| 95 | G4_Am | 13.67 | 934 |
| 96 | G4_Cm | 13.51 | 939 |
| 97 | G4_Bk | 14 | 952 |
| 98 | G4_Cf | 10 | 966 |

### 11.1.2 NIST Compounds

```
=============================================================
 Ncomp             Name      density(g/cm^3)  I(eV) ChFormula
=============================================================
 6            G4_A-150_TISSUE      1.127      65.1
        1      0.101327
        6        0.7755
        7      0.035057
        8     0.0523159
        9      0.017422
       20      0.018378
 3                 G4_ACETONE     0.7899      64.2
        6             3
        1             6
        8             1
 2              G4_ACETYLENE  0.0010967      58.2
        6             2
        1             2
 3                 G4_ADENINE        1.6      71.4
        6             5
        1             5
        7             5
 7      G4_ADIPOSE_TISSUE_ICRP       0.95      63.2
        1         0.114
        6         0.598
        7         0.007
        8         0.278
       11         0.001
       16         0.001
       17         0.001
 4                     G4_AIR 0.00120479      85.7
        6      0.000124
        7      0.755268
        8      0.231781
       18      0.012827
```

(continues on next page)

(continued from previous page)

```
 4               G4_ALANINE       1.42      71.9
         6             3
         1             7
         7             1
         8             2
 2          G4_ALUMINUM_OXIDE       3.97     145.2   Al_2O_3
        13             2
         8             3
 3                 G4_AMBER        1.1      63.2
         1       0.10593
         6      0.788974
         8      0.105096
 2               G4_AMMONIA 0.000826019      53.7
         7             1
         1             3
 3               G4_ANILINE     1.0235      66.2
         6             6
         1             7
         7             1
 2            G4_ANTHRACENE      1.283      69.5
         6            14
         1            10
 6            G4_B-100_BONE       1.45      85.9
         1     0.0654709
         6      0.536944
         7        0.0215
         8      0.032085
         9      0.167411
        20      0.176589
 3              G4_BAKELITE       1.25      72.4
         1      0.057441
         6      0.774591
         8      0.167968
 2        G4_BARIUM_FLUORIDE       4.89     375.9
        56             1
         9             2
 3         G4_BARIUM_SULFATE        4.5     285.7
        56             1
        16             1
         8             4
 2               G4_BENZENE    0.87865      63.4
         6             6
         1             6
 2        G4_BERYLLIUM_OXIDE       3.01      93.2
         4             1
         8             1
 3                   G4_BGO       7.13     534.1
        83             4
        32             3
         8            12
10            G4_BLOOD_ICRP       1.06      75.2
         1         0.102
         6          0.11
         7         0.033
         8         0.745
        11         0.001
        15         0.001
        16         0.002
        17         0.003
        19         0.002
        26         0.001
 8       G4_BONE_COMPACT_ICRU       1.85      91.9
         1         0.064
         6         0.278
         7         0.027
         8          0.41
```

(continues on next page)

(continued from previous page)

```
        12         0.002
        15          0.07
        16         0.002
        20         0.147
9       G4_BONE_CORTICAL_ICRP       1.92        110
         1         0.034
         6         0.155
         7         0.042
         8         0.435
        11         0.001
        12         0.002
        15         0.103
        16         0.003
        20         0.225
2            G4_BORON_CARBIDE       2.52       84.7
         5             4
         6             1
2              G4_BORON_OXIDE      1.812       99.6
         5             2
         8             3
9               G4_BRAIN_ICRP       1.04       73.3
         1         0.107
         6         0.145
         7         0.022
         8         0.712
        11         0.002
        15         0.004
        16         0.002
        17         0.003
        19         0.003
2                   G4_BUTANE 0.00249343       48.3
         6             4
         1            10
3          G4_N-BUTYL_ALCOHOL     0.8098       59.9
         6             4
         1            10
         8             1
5                    G4_C-552       1.76       86.8
         1       0.02468
         6      0.501611
         8      0.004527
         9      0.465209
        14      0.003973
2        G4_CADMIUM_TELLURIDE        6.2      539.3
        48             1
        52             1
3        G4_CADMIUM_TUNGSTATE        7.9      468.3
        48             1
        74             1
         8             4
3        G4_CALCIUM_CARBONATE        2.8      136.4
        20             1
         6             1
         8             3
2         G4_CALCIUM_FLUORIDE       3.18        166
        20             1
         9             2
2            G4_CALCIUM_OXIDE        3.3      176.1
        20             1
         8             1
3          G4_CALCIUM_SULFATE       2.96      152.3
        20             1
        16             1
         8             4
3        G4_CALCIUM_TUNGSTATE      6.062        395
        20             1
```

(continues on next page)

(continued from previous page)

```
       74             1
        8             4
 2          G4_CARBON_DIOXIDE 0.00184212         85    CO_2
        6             1
        8             2
 2    G4_CARBON_TETRACHLORIDE      1.594      166.3
        6             1
       17             4
 3    G4_CELLULOSE_CELLOPHANE       1.42       77.6
        6             6
        1            10
        8             5
 3      G4_CELLULOSE_BUTYRATE        1.2       74.6
        1      0.067125
        6      0.545403
        8      0.387472
 4       G4_CELLULOSE_NITRATE       1.49         87
        1      0.029216
        6      0.271296
        7      0.121276
        8      0.578212
 5           G4_CERIC_SULFATE       1.03       76.7
        1      0.107596
        7        0.0008
        8      0.874976
       16      0.014627
       58      0.002001
 2         G4_CESIUM_FLUORIDE      4.115      440.7
       55             1
        9             1
 2           G4_CESIUM_IODIDE       4.51      553.1
       55             1
       53             1
 3           G4_CHLOROBENZENE     1.1058       89.1
        6             6
        1             5
       17             1
 3              G4_CHLOROFORM     1.4832        156
        6             1
        1             1
       17             3
10                G4_CONCRETE        2.3      135.2
        1          0.01
        6         0.001
        8      0.529107
       11         0.016
       12         0.002
       13      0.033872
       14      0.337021
       19         0.013
       20         0.044
       26         0.014
 2             G4_CYCLOHEXANE      0.779       56.4
        6             6
        1            12
 3     G4_1,2-DICHLOROBENZENE     1.3048      106.5
        6             6
        1             4
       17             2
 4   G4_DICHLORODIETHYL_ETHER     1.2199      103.3
        6             4
        1             8
        8             1
       17             2
 3      G4_1,2-DICHLOROETHANE     1.2351      111.9
        6             2
```

(continues on next page)

(continued from previous page)

```
        1            4
       17            2
3          G4_DIETHYL_ETHER    0.71378        60
        6            4
        1           10
        8            1
4  G4_N,N-DIMETHYL_FORMAMIDE     0.9487      66.6
        6            3
        1            7
        7            1
        8            1
4       G4_DIMETHYL_SULFOXIDE     1.1014      98.6
        6            2
        1            6
        8            1
       16            1
2                 G4_ETHANE 0.00125324      45.4
        6            2
        1            6
3          G4_ETHYL_ALCOHOL     0.7893      62.9
        6            2
        1            6
        8            1
3        G4_ETHYL_CELLULOSE       1.13      69.3
        1      0.090027
        6      0.585182
        8      0.324791
2               G4_ETHYLENE 0.00117497      50.7
        6            2
        1            4
8          G4_EYE_LENS_ICRP       1.07      73.3
        1         0.096
        6         0.195
        7         0.057
        8         0.646
       11         0.001
       15         0.001
       16         0.003
       17         0.001
2           G4_FERRIC_OXIDE        5.2     227.3
       26            2
        8            3
2            G4_FERROBORIDE       7.15       261
       26            1
        5            1
2          G4_FERROUS_OXIDE        5.7     248.6
       26            1
        8            1
7        G4_FERROUS_SULFATE      1.024      76.4
        1      0.108259
        7       2.7e-05
        8      0.878636
       11       2.2e-05
       16      0.012968
       17       3.4e-05
       26       5.4e-05
3               G4_FREON-12       1.12       143
        6      0.099335
        9      0.314247
       17      0.586418
3             G4_FREON-12B2        1.8     284.9
        6      0.057245
        9      0.181096
       35      0.761659
3               G4_FREON-13       0.95     126.6
        6      0.114983
```

(continues on next page)

(continued from previous page)

```
        9      0.545621
       17      0.339396
3              G4_FREON-13B1        1.5     210.5
        6             1
        9             3
       35             1
3              G4_FREON-13I1        1.8     293.5
        6      0.061309
        9      0.290924
       53      0.647767
3   G4_GADOLINIUM_OXYSULFIDE       7.44     493.3
       64             2
        8             2
       16             1
2        G4_GALLIUM_ARSENIDE       5.31     384.9
       31             1
       33             1
5      G4_GEL_PHOTO_EMULSION     1.2914      74.8
        1       0.08118
        6       0.41606
        7       0.11124
        8       0.38064
       16       0.01088
6              G4_Pyrex_Glass       2.23       134
        5     0.0400639
        8      0.539561
       11     0.0281909
       13      0.011644
       14      0.377219
       19    0.00332099
5              G4_GLASS_LEAD       6.22     526.4
        8      0.156453
       14      0.080866
       22      0.008092
       33      0.002651
       82      0.751938
4              G4_GLASS_PLATE        2.4     145.4
        8        0.4598
       11     0.0964411
       14      0.336553
       20      0.107205
4               G4_GLUTAMINE       1.46      73.3
        6             5
        1            10
        7             2
        8             3
3                G4_GLYCEROL     1.2613      72.6
        6             3
        1             8
        8             3
4                 G4_GUANINE        2.2        75
        6             5
        1             5
        7             5
        8             1
4                  G4_GYPSUM       2.32     129.7
       20             1
       16             1
        8             6
        1             4
2               G4_N-HEPTANE    0.68376      54.4
        6             7
        1            16
2                G4_N-HEXANE     0.6603        54
        6             6
        1            14
```

(continues on next page)

(continued from previous page)

```
4                 G4_KAPTON       1.42       79.6
        6            22
        1            10
        7             2
        8             5
3    G4_LANTHANUM_OXYBROMIDE       6.28      439.7
       57             1
       35             1
        8             1
3    G4_LANTHANUM_OXYSULFIDE       5.86      421.2
       57             2
        8             2
       16             1
2             G4_LEAD_OXIDE       9.53      766.7
        8      0.071682
       82      0.928318
3          G4_LITHIUM_AMIDE      1.178       55.5
        3             1
        7             1
        1             2
3       G4_LITHIUM_CARBONATE       2.11       87.9
        3             2
        6             1
        8             3
2        G4_LITHIUM_FLUORIDE      2.635         94
        3             1
        9             1
2         G4_LITHIUM_HYDRIDE       0.82       36.5
        3             1
        1             1
2          G4_LITHIUM_IODIDE      3.494      485.1
        3             1
       53             1
2           G4_LITHIUM_OXIDE      2.013       73.6
        3             2
        8             1
3      G4_LITHIUM_TETRABORATE       2.44       94.6
        3             2
        5             4
        8             7
9               G4_LUNG_ICRP       1.04       75.3
        1         0.105
        6         0.083
        7         0.023
        8         0.779
       11         0.002
       15         0.001
       16         0.002
       17         0.003
       19         0.002
5                 G4_M3_WAX       1.05       67.9
        1      0.114318
        6      0.655824
        8     0.0921831
       12      0.134792
       20      0.002883
3      G4_MAGNESIUM_CARBONATE      2.958        118
       12             1
        6             1
        8             3
2      G4_MAGNESIUM_FLUORIDE          3      134.3
       12             1
        9             2
2         G4_MAGNESIUM_OXIDE       3.58      143.8
       12             1
        8             1
```

(continues on next page)

(continued from previous page)

```
3   G4_MAGNESIUM_TETRABORATE       2.53     108.3
       12             1
        5             4
        8             7
2         G4_MERCURIC_IODIDE       6.36     684.5
       80             1
       53             2
2                 G4_METHANE 0.000667151      41.7
        6             1
        1             4
3                G4_METHANOL     0.7914      67.6
        6             1
        1             4
        8             1
5               G4_MIX_D_WAX       0.99      60.9
        1       0.13404
        6       0.77796
        8       0.03502
       12      0.038594
       22      0.014386
6             G4_MS20_TISSUE          1      75.1
        1      0.081192
        6      0.583442
        7      0.017798
        8      0.186381
       12      0.130287
       17        0.0009
9    G4_MUSCLE_SKELETAL_ICRP       1.05      75.3
        1         0.102
        6         0.143
        7         0.034
        8          0.71
       11         0.001
       15         0.002
       16         0.003
       17         0.001
       19         0.004
8    G4_MUSCLE_STRIATED_ICRU       1.04      74.7
        1      0.102102
        6      0.123123
        7      0.035035
        8       0.72973
       11      0.001001
       15      0.002002
       16      0.004004
       19      0.003003
4     G4_MUSCLE_WITH_SUCROSE       1.11      74.3
        1     0.0982341
        6      0.156214
        7      0.035451
        8      0.710101
4  G4_MUSCLE_WITHOUT_SUCROSE       1.07      74.2
        1      0.101969
        6      0.120058
        7      0.035451
        8      0.742522
2             G4_NAPHTHALENE      1.145      68.4
        6            10
        1             8
4            G4_NITROBENZENE    1.19867      75.8
        6             6
        1             5
        7             1
        8             2
2           G4_NITROUS_OXIDE 0.00183094      84.9
        7             2
```

(continues on next page)

(continued from previous page)

```
        8           1
4              G4_NYLON-8062       1.08      64.3
        1      0.103509
        6      0.648416
        7     0.0995361
        8      0.148539
4               G4_NYLON-6-6       1.14      63.9
        6           6
        1          11
        7           1
        8           1
4              G4_NYLON-6-10       1.14      63.2
        1      0.107062
        6      0.680449
        7      0.099189
        8        0.1133
4         G4_NYLON-11_RILSAN      1.425      61.6
        1      0.115476
        6      0.720818
        7     0.0764169
        8     0.0872889
2                  G4_OCTANE     0.7026      54.7
        6           8
        1          18
2                G4_PARAFFIN       0.93      55.9
        6          25
        1          52
2               G4_N-PENTANE     0.6262      53.6
        6           5
        1          12
8          G4_PHOTO_EMULSION      3.815       331
        1        0.0141
        6      0.072261
        7       0.01932
        8      0.066101
       16       0.00189
       35      0.349103
       47      0.474105
       53       0.00312
2 G4_PLASTIC_SC_VINYLTOLUENE      1.032      64.7
        6           9
        1          10
2       G4_PLUTONIUM_DIOXIDE      11.46     746.5
       94           1
        8           2
3       G4_POLYACRYLONITRILE       1.17      69.6
        6           3
        1           3
        7           1
3           G4_POLYCARBONATE        1.2      73.1
        6          16
        1          14
        8           3
3       G4_POLYCHLOROSTYRENE        1.3      81.7
        6           8
        1           7
       17           1
2             G4_POLYETHYLENE       0.94      57.4   (C_2H_4)_N-Polyethylene
        6           1
        1           2
3                   G4_MYLAR        1.4      78.7
        6          10
        1           8
        8           4
3              G4_PLEXIGLASS       1.19        74
        6           5
```

(continues on next page)

(continued from previous page)

```
        1             8
        8             2
3        G4_POLYOXYMETHYLENE      1.425      77.4
        6             1
        1             2
        8             1
2           G4_POLYPROPYLENE        0.9      56.5   (C_2H_4)_N-Polypropylene
        6             2
        1             4
2             G4_POLYSTYRENE       1.06      68.7
        6             8
        1             8
2                  G4_TEFLON        2.2      99.1
        6             2
        9             4
3 G4_POLYTRIFLUOROCHLOROETHYLENE        2.1     120.7
        6             2
        9             3
       17             1
3       G4_POLYVINYL_ACETATE       1.19      73.7
        6             4
        1             6
        8             2
3       G4_POLYVINYL_ALCOHOL        1.3      69.7
        6             2
        1             4
        8             1
3       G4_POLYVINYL_BUTYRAL       1.12      67.2
        6             8
        1            14
        8             2
3      G4_POLYVINYL_CHLORIDE        1.3     108.2
        6             2
        1             3
       17             1
3 G4_POLYVINYLIDENE_CHLORIDE        1.7     134.3
        6             2
        1             2
       17             2
3 G4_POLYVINYLIDENE_FLUORIDE       1.76      88.8
        6             2
        1             2
        9             2
4   G4_POLYVINYL_PYRROLIDONE       1.25      67.7
        6             6
        1             9
        7             1
        8             1
2        G4_POTASSIUM_IODIDE       3.13     431.9
       19             1
       53             1
2         G4_POTASSIUM_OXIDE       2.32     189.9
       19             2
        8             1
2                 G4_PROPANE 0.00187939      47.1
        6             3
        1             8
2                G4_lPROPANE       0.43        52
        6             3
        1             8
3        G4_N-PROPYL_ALCOHOL     0.8035      61.1
        6             3
        1             8
        8             1
3                G4_PYRIDINE     0.9819      66.2
        6             5
```

(continues on next page)

(continued from previous page)

```
        1            5
        7            1
2          G4_RUBBER_BUTYL       0.92      56.5
        1      0.143711
        6      0.856289
2        G4_RUBBER_NATURAL       0.92      59.8
        1      0.118371
        6      0.881629
3       G4_RUBBER_NEOPRENE       1.23        93
        1       0.05692
        6      0.542646
       17      0.400434
2       G4_SILICON_DIOXIDE       2.32     139.2   SiO_2
       14            1
        8            2
2        G4_SILVER_BROMIDE      6.473     486.6
       47            1
       35            1
2       G4_SILVER_CHLORIDE       5.56     398.4
       47            1
       17            1
3        G4_SILVER_HALIDES       6.47     487.1
       35      0.422895
       47      0.573748
       53      0.003357
2         G4_SILVER_IODIDE       6.01     543.5
       47            1
       53            1
9             G4_SKIN_ICRP       1.09      72.7
        1          0.1
        6        0.204
        7        0.042
        8        0.645
       11        0.002
       15        0.001
       16        0.002
       17        0.003
       19        0.001
3      G4_SODIUM_CARBONATE      2.532       125
       11            2
        6            1
        8            3
2         G4_SODIUM_IODIDE      3.667       452
       11            1
       53            1
2       G4_SODIUM_MONOXIDE       2.27     148.8
       11            2
        8            1
3        G4_SODIUM_NITRATE      2.261     114.6
       11            1
        7            1
        8            3
2              G4_STILBENE     0.9707      67.7
        6           14
        1           12
3               G4_SUCROSE     1.5805      77.5
        6           12
        1           22
        8           11
2             G4_TERPHENYL       1.24      71.7
        6           18
        1           14
9           G4_TESTIS_ICRP       1.04        75
        1        0.106
        6        0.099
        7         0.02
```

(continues on next page)

(continued from previous page)

```
       8         0.766
      11         0.002
      15         0.001
      16         0.002
      17         0.002
      19         0.002
2     G4_TETRACHLOROETHYLENE      1.625     159.2
       6             2
      17             4
2       G4_THALLIUM_CHLORIDE      7.004     690.3
      81             1
      17             1
9         G4_TISSUE_SOFT_ICRP       1.03      72.3
       1         0.105
       6         0.256
       7         0.027
       8         0.602
      11         0.001
      15         0.002
      16         0.003
      17         0.002
      19         0.002
4      G4_TISSUE_SOFT_ICRU-4          1      74.9
       1         0.101
       6         0.111
       7         0.026
       8         0.762
4          G4_TISSUE-METHANE 0.00106409      61.2
       1      0.101869
       6      0.456179
       7      0.035172
       8       0.40678
4          G4_TISSUE-PROPANE 0.00182628      59.5
       1      0.102672
       6       0.56894
       7      0.035022
       8      0.293366
2         G4_TITANIUM_DIOXIDE       4.26     179.5
      22             1
       8             2
2                 G4_TOLUENE     0.8669      62.5
       6             7
       1             8
3       G4_TRICHLOROETHYLENE       1.46     148.1
       6             2
       1             1
      17             3
4      G4_TRIETHYL_PHOSPHATE       1.07      81.2
       6             6
       1            15
       8             4
      15             1
2   G4_TUNGSTEN_HEXAFLUORIDE        2.4     354.4
      74             1
       9             6
2       G4_URANIUM_DICARBIDE      11.28       752
      92             1
       6             2
2     G4_URANIUM_MONOCARBIDE      13.63       862
      92             1
       6             1
2           G4_URANIUM_OXIDE      10.96     720.6
      92             1
       8             2
4                    G4_UREA      1.323      72.8
       6             1
```

(continues on next page)

(continued from previous page)

```
        1             4
        7             2
        8             1
4                 G4_VALINE       1.23      67.7
        6             5
        1            11
        7             1
        8             2
3                  G4_VITON        1.8      98.6
        1      0.009417
        6      0.280555
        9      0.710028
2                  G4_WATER          1        78   H_2O
        1             2
        8             1
2            G4_WATER_VAPOR 0.000756182       71.6   H_2O-Gas
        1             2
        8             1
2                 G4_XYLENE       0.87      61.8
        6             8
        1            10
1               G4_GRAPHITE       2.21        78   Graphite
```

### 11.1.3 HEP and Nuclear Materials

```
=============================================================
 Ncomp                Name      density(g/cm^3)  I(eV) ChFormula
=============================================================
 1                      G4_lH2     0.0708      21.8
 1                      G4_lN2      0.807        82
 1                      G4_lO2      1.141        95
 1                      G4_lAr      1.396       188
 1                      G4_lBr     3.1028       343
 1                      G4_lKr      2.418       352
 1                      G4_lXe      2.953       482
 3                    G4_PbWO4       8.28         0
         8             4
        82             1
        74             1
 1                 G4_Galactic      1e-25      21.8
 1          G4_GRAPHITE_POROUS        1.7        78   Graphite
 3                   G4_LUCITE       1.19        74
         1      0.080538
         6      0.599848
         8      0.319614
 3                    G4_BRASS       8.52         0
        29            62
        30            35
        82             3
 3                   G4_BRONZE       8.82         0
        29            89
        30             9
        82             2
 3          G4_STAINLESS-STEEL          8         0
        26            74
        24            18
        28             8
 3                     G4_CR39       1.32         0
         1            18
         6            12
         8             7
 3              G4_OCTADECANOL      0.812         0
         1            38
```

(continues on next page)

(continued from previous page)

```
         6             18
         8              1
```

## 11.1.4 Space (ISS) Materials

```
=============================================================
 Ncomp              Name      density(g/cm^3)  I(eV)  ChFormula
=============================================================
 4                 G4_KEVLAR       1.44         0
         6             14
         1             10
         8              2
         7              2
 3                 G4_DACRON        1.4         0
         6             10
         1              8
         8              4
 3               G4_NEOPRENE       1.23         0
         6              4
         1              5
        17              1
```

## 11.1.5 Bio-Chemical Materials

```
=============================================================
 Ncomp              Name      density(g/cm^3)  I(eV)  ChFormula
=============================================================
 4               G4_CYTOSINE       1.55        72
         1              5
         6              4
         7              3
         8              1
 4                G4_THYMINE       1.23        72
         1              6
         6              5
         7              2
         8              2
 4                 G4_URACIL       1.32        72
         1              4
         6              4
         7              2
         8              2
 3            G4_DNA_ADENINE          1        72
         1              4
         6              5
         7              5
 4            G4_DNA_GUANINE          1        72
         1              4
         6              5
         7              5
         8              1
 4           G4_DNA_CYTOSINE          1        72
         1              4
         6              4
         7              3
         8              1
 4            G4_DNA_THYMINE          1        72
         1              5
         6              5
         7              2
```

(continues on next page)

(continued from previous page)

```
        8             2
4              G4_DNA_URACIL          1        72
        1             3
        6             4
        7             2
        8             2
4           G4_DNA_ADENOSINE          1        72
        1            10
        6            10
        7             5
        8             4
4           G4_DNA_GUANOSINE          1        72
        1            10
        6            10
        7             5
        8             5
4            G4_DNA_CYTIDINE          1        72
        1            10
        6             9
        7             3
        8             5
4             G4_DNA_URIDINE          1        72
        1             9
        6             9
        7             2
        8             6
4       G4_DNA_METHYLURIDINE          1        72
        1            11
        6            10
        7             2
        8             6
2       G4_DNA_MONOPHOSPHATE          1        72
       15             1
        8             3
5                   G4_DNA_A          1        72
        1            10
        6            10
        7             5
        8             7
       15             1
5                   G4_DNA_G          1        72
        1            10
        6            10
        7             5
        8             8
       15             1
5                   G4_DNA_C          1        72
        1            10
        6             9
        7             3
        8             8
       15             1
5                   G4_DNA_U          1        72
        1             9
        6             9
        7             2
        8             9
       15             1
5                  G4_DNA_MU          1        72
        1            11
        6            10
        7             2
        8             9
       15             1
```

## 11.2 Transportation in Magnetic Field - Further Details

### 11.2.1 The challenge of integrating all tracks

**What leads us to discard tracks looping in a magnetic field**

The integration of charged particle tracks in magnetic field is an important part of the computational cost (CPU time). Part of this cost is due to integration of low-energy particles in a volume with low density and strong magnetic field.

In HEP applications the most important type of tracks causing such problems are electrons in the vacuum of beam pipes. Charged particles in volumes near decay volumes and muons in large volumes of air are other examples.

To limit this CPU cost, a type of tracking cut for charged particles was introduced in Geant4 release 7.0 in `G4Transportation` and `G4CoupledTransportation`. Tracks which require more than a threshold number of integration steps [maxLoopCount] (currently 1,000) during a physics/tracking step are marked as 'looping' and are considered candidates for being killed - i.e. they can potentially be abandoned after the current step, and have their energy deposited locally.

Enhancements introduced in release 10.6 provide more comprehensive information about the tracks killed, in the form of `G4Exception` warning messages.

This section describes this policy, the parameters which the user is able to set to tune it, and recent refinements implemented in Geant4 10.5.

**Cost of integration**

Occasionally tracks 'looping' in a strong magnetic field, making little progress even over thousands of integration steps. This is due to a combination of a strong magnetic field and a thin material (gas or vacuum) in which the size of a physics step is substantially larger than the radius of curvature of the track.

The preferred integration method for tracks in an EM field is the Runge-Kutta method. This and other similar methods are well suited to variations in magnetic fields and step sizes up to a few times the radius of curvature of the charged tracks.

However when the step sizes are hundreds or thousands of times larger than the curvature of the track, these methods are expensive as they do not progress the integration of a track adequately.

The amount of CPU time which can be consumed by one or few such tracks can very large, sometimes contributing per cent increases to the simulation of some primary particles. Some tracks with a very small drift velocity (projection of the velocity along the vector of the magnetic field) can stop the progress of a simulation if they are not limited or integrated using alternative means.

So it is important to limit the number of integration steps spent on these tracks. The module for propagation in field in Geant4 flags tracks which take more than a certain number (default 1,000) integration steps without reaching the requested end of the step size, which was determined by the physics and geometry.

**Parameters for eliminating or controling which particles are killed**

The Geant4 `G4Transportation` and `G4CoupledTransportation` processes are tasked to select which of the tracks flagged as looping are killed and which survive. To balance the potential significant cost of integrating looping particles, three thresholds exist: - the 'Warning' Energy: a track with energy below this value that is found to loop is killed silently (no warning.) Above the 'Warning Energy', if a track is selected for killing then a warning is generated. - the 'Important' Energy: the threshold energy above which a track will survive for multiple steps if found looping. - the number of extra 'tracking' steps for important particles. These tracks will be only be killed only if they still loop more than this number of trial steps. ( So in effect the number of integration steps will be this number times the maximum number of steps allowed in `G4PropagatorInField`.

In versions of Geant4 from 7.0 up to release 10.4, Transportation did not examine the types of a charged particle - all types of particles were killed if they fulfilled the same criteria. A short message was written in the `G4cout` output that gave the energy and location of the killed track. This printout was under the `G4VERBOSE` flag, so it was suppressed if the `G4_NO_VERBOSE` configuration option was chosen at installation.

In Geant4 10.5 several changes have been implemented:

- only stable particles are killed. ( Re-enabling the killing of unstable particles as an option is envisioned. )
- each particle with energy above the warning energy which is killed generates a detailed warning (using `G4Exception`) with the full information about the particle location, the current volume and its material, and the particle momentum and energy.
- for the first 5 tracks killed a detailed description is printed that describes the criteria and parameters which are used to decide what tracks are killed, and provides a first guidance regarding how to 'save' tracks by chaning the values of thresholds or by adopting different integration methods.

Below we discuss the different way in which a user can change the thresholds for killing 'looping' tracks, which criteria can be used to ensure that a track continues to propagate and for how many steps an 'important' track that is 'looping' can survive.

Two techniques are demonstrated below. An example of using them is available in the extended example `field01`, in the directory `examples/extended/field/field01`.

## 11.2.2 Using preset thresholds for killing loopers

This method is new in Geant4 release 10.5, and uses the G4PhysicsListHelper which has methods to choose a preselected set of parameter values. The choices are between a set each of low and high thresholds. Either one can be enabled by calling correspondingly method.

It is possible to select a set of pre-selected values of the parameters for looping particles using

New functionality in `G4PhysicsListHelper`, introduced in Geant4 release 10.5, enables the Transportation process chosen to be provided with this set of parameters. This reuses the `AddTransportation` method, which is called in each thread.

To configure with low values of the thresholds, appropriate for typical applications using low-energy physics, choose

```
#include "G4PhysicsListHelper.hh"

int main( int, char**)
{
  auto plHelper = G4PhysicsListHelper::GetPhysicsListHelper();

  plHelper->UseLowLooperThresholds();
  //
  // Use low values for the thresholds
  //    Warning  1 keV, Important 1 MeV, trials 10

  auto physList = new FTFP_BERT();
  // ...
}
```

The original high values of the parameters can be selected with a similar call

```
plHelper->UseHighLooperThresholds();
//
// Configures with the original (high) values of parameters. Currently:
//    Warning 100 MeV, Important 250 MeV, trials 10
```

and are chosen as starting points for energy-frontier HEP experiments.

The above sequence is demonstrated in the `main()` of `field01.cc`, part of the extended field examples ( `examples/extended/field/field01` .)

This configuration method works only if modular physics lists are used, or if the AddTransportation() method is used to construct the transportation in a user physics list.

These calls must be done *before* a physics list is instantiated, in particular before `G4PhysicsListHelper::AddTransportation()` is called during the construction of a physics list. Else the configuration of the parameters does not occur.

These methods must be called before the physics is constructed - i.e. typically before `G4RunManager` 's `Initialise` method is called.

In order for this method to work the physics list must be constructed in one of two ways:

- a preconstructed physics lists, from the list of recommended physics lists, or
- the list must be constructed using the `G4ModularPhysicsList` and its `AddTransportation` method.

Note that in each thread the `AddTransportation` instantiates a single common transportation process which is then used by all particles types.

Users who build a physics list without making use of `G4ModularPhysicsList` and its `AddTransportation` method, are responsible to register a `Transportation` process to each particle type, and to set its parameters appropriately. This would allow the most finer grained control, and would also allow different thresholds to be chosen for different particle types.

### 11.2.3 Finer-grain control of the parameters for killing looping particles

A new feature to set any value to these parameters is introduced in Geant4 release 11.1 using the class `G4TransportationParameters`. If an instance of `G4TransportationParameters` exists, the constructor of `G4Transportation` will utilise the values it stored to inititalise its own parameters.

```
auto transportParams= G4TransportationParameters::Instance();

transportParams->SetWarningEnergy(  warningE );
transportParams->SetImportantEnergy( importantE );
transportParams->SetNumberOfTrials( numTrials );
G4cout << "Using G4TransportationParameters to set looper parameters."  << G4endl;
```

A couple of caveats exist. First is that its values will be used by default for all instances of *G4Transporation* and its derived classes: whether it is

- the single instance typically registered for all particles (as is done in the modular physics lists), or
- an instance created separately and registered by a user to one or more (charged) particles as a replacement.

Secondly, if it exists, the values of all the parameters that `G4TransportationParameters` stores are currently used to overwrite the existing default values in `G4Transportation`.

So, if you create `G4TransportationParameters` it is your responsibility to set the values of all of its parameters.

### 11.2.4 Full control of the parameters for killing looping particles

Whether you use one of the previous methods or not, it is possible to exercise full fine-grained control over each values for each type of particle separately.

The user can choose arbitrary values for the different parametes related to killing loopers and also refine the integration of charged particle propagation in particular volumes in order to eliminate or reduce the incidence of looping tracks.

This is also the only method which will work in all Geant4 versions since 7.0.

To obtain reliable configuration of the `G4Transportation` (or `G4CoupledTransportation` ) process in a potentially multi-threaded application, we configure it using a `G4VUserRunAction`. In particular such configuration can be undertaken in the `BeginOfRunAction` methods.

For example, to ensure that only looping particles with energy 10 keV are killed silently we change the value of the 'Warning' Energy

```
transport->SetThresholdWarningEnergy( 1.0 * CLHEP::keV  );
```

After this each time a (stable) looping track with energy over 1.0 keV is killed by this transportation process, it will generate a warning.

The second configurable energy threshold is labelled the 'important' energy and it enables tracks above its value to survive a chosen number of 'tracking' steps. They will be only be killed only if they are still looping after the given number of tracking steps.

These are demonstrated also in the `F01RunAction`'s `ChangeLooperParameters` method, which is called by the `BeginOfRunAction`.

To obtain the appropriate Transportation object for a particular particle type `G4ParticleDefinition *particleDef;` either obtain it manually obtain directly if we know its type

```
G4VProcess* partclTransport =
 particleDef->GetProcessManager()->GetProcess("G4Transportation");

auto transport= dynamic_cast<G4Transportation*>(partclTransport);
```

or write code which can adapt to different types to different types which inherit from `G4Transportation`

Listing 11.1: A method to find the transportation object for a particle type. It can also find the ordinary `G4Transportation` or the derived classes `G4CoupledTransportation` and the newest `G4TransportationWithMsc`.

```
G4Transportation* FindTransport( G4ParticleDefinition* particleDef )
{
  G4Transportation *transport;
  G4VProcess* cplTransport= nullptr, *transportMsc= nullptr;

  auto pm= particleDef->GetProcessManager();

  G4VProcess* ordTransport = pm->GetProcess("G4Transportation");
  transport= dynamic_cast<G4Transportation*>(ordTransport);

  if( !transport ) {
    // Maybe it is G4CoupledTransportation ...
    cplTransport= pm->GetProcess("G4CoupledTransportation");
    if ( vpCoupledTransport) {
      transport= dynamic_cast<G4Transportation*>(cplTransport);
    }
  }
  return transport;
}
```

or else use (or copy) the helper method `F01RunAction::FindTransportation`

```
auto transport= FindTransportation(G4Electron::Definition(), true);
```

This example method returns returns a ``G4Transportation *``. ( whereas in release 11.0 it returned a pair ``std::pair<G4Transportation*, G4CoupledTransportation*>``. )

Since Geant4 11.1, `G4CoupledTransportation` and the new `G4TransportationWithMsc` classes inherit from `G4Transportation` you can use common code to configure them (as shown in Listing 11.1).

In case a different Transportation type is used which does not inherit from `G4Transportation`, such as `G4ITTransportation`, and `G4DNABrownianTransportation` (both relevant for Geant4 DNA) similar code is required for each such class.

```
auto pm= particleDef->GetProcessManager();
G4VProcess* vpItTransport= pm->GetProcess("G4ITTransportation");
auto itTransport= dynamic_cast<G4ITTransportation*>(vpItTransport);
```

NOTE: Up to (and including) release 11.0 `G4CoupledTransportation` was an independent class, not inheriting from `G4Transportation`. If your application needs to be backward compatible with previous releases of Geant4 (including release 10.1 through 10.7 and 11.0) you must ignore this new inheritance relationship.

When using earlier version of Geant4 it was necessary to treat instances of `G4CoupledTransportation` separately:

Listing 11.2: Demonstration of applying cuts to an instance of `G4CoupledTransportation`.

```
G4VProcess* vProcCoupled= pm->GetProcess("G4CoupledTransportation");
G4CoupledTransportation* coupledTransport=
     dynamic_cast<G4CoupledTransportation*>(vProcCoupled);

coupledTransport->SetThresholdWarningEnergy( 1.0 * CLHEP::keV  );
coupledTransport->SetThresholdImportantEnergy( 1.0 * CLHEP::MeV );
coupledTransport->SetNumberOfTrials( 20 );
```

It is still possible to use this approach (listing Listing 11.2) if you are maintaining an application which must work both older versions and the current release 11.1.

However moving forward the code can now be simplified, as demonstrated in the next two code excerpts. First by obtaining a `G4Transportation``, e.g. as in listing Listing 11.1.

Then using common code, as in listing Listing 11.3 to overwrite the thresholds in a `G4Transportation` (or derived) class as found in `F01RunAction` 's `ChangeLooperParameters` method

Listing 11.3: Adapted extract of the method `ChangeLooperParameters` from `F01RunAction`

```
void ChangeLooperParameters(const G4ParticleDefinition* particleDef )
{
  auto transport= FindTransportation( particleDef );

  // Since Geant4 11.1 the following code works for several transportation
  //  classes:
  //   - ordinary G4Transporation,
  //   - G4CoupledTransportation used for parallel worlds, and
  //   - G4TransportationWithMsc used to speed up charged particles.

  if( transport != nullptr )
  {
    // Change the values of the looping particle parameters of Transportation
    transport->SetThresholdWarningEnergy(  warningEnergy );
    transport->SetThresholdImportantEnergy(  importantEnergy );
    transport->SetThresholdTrials( numberOfTrials );
  }
}
```

Note that for all pre-configured and modular physics lists share a single Transportation process for all types of particles. So the parameters for killing loopers will be shared by all particle types in this case.

If this is not the desired behaviour, it is necessary to register a separate instance of the Transporation process for a particular type of particle. See the subsection *ReplacingTransportation* about how this can be done.

`F01RunAction` plays the role of a helper object, which holds the proposed (new) values of parameters, and which can allow them to be set, e.g., in the `main()` function

```
runAction->SetWarningEnergy(   10.0 * CLHEP::keV );
```

`F01RunAction` then forwards them to the `Transportation` object of each thread at the start of each run.

### 11.2.5 Using a helper object to forward parameter changes

Since the type of the transportation it can be useful to use a helper object to hold the desired values for the parameters (thresholds, number of iterations), and to forward them to the `Transportation` class.

This is demonstrated in the class `F01RunAction` and its `ChangeLooperParameters` method of the field01 extended example.

It copes with either transportation class, `G4Transportation` or a `G4CoupledTransportation`, and passes new values of parameters as needed.

In `field01` the methods of F01RunAction

```
runAction->SetImportantEnergy( 0.1  * CLHEP::MeV );
runAction->SetNumberOfTrials( 30 );
```

which the run action passes to the `G4Transportation` or `G4CoupledTransportation` object registered for the electron in `F01RunAction` ‘s method `ChangeLooperParameters`.

### 11.2.6 How to replace the Transportation Process of a particle type

The most advanced use case of controling requires a separate instance of a *G4Transporation* (or *G4CoupledTransporation* or other process).

If you have configured your application to use *G4TransporationWithMsc* only for electrons

Currently in order to undertake this it is necessary to use the property of the transportation of being first in the process list and interact directly with the process manager:

Listing 11.4: Replacing the G4Transportation process for one particle type - electrons

```
G4ParticleDefinition* particleDef= G4Electron::Definition();
G4int verboseLevel= 0;

G4ProcessManager* procManager = particleDef->GetProcessManager();
auto plist = procManager->GetProcessList();

procManager->RemoveProcess(0);  // Remove the current Transport

auto transport = new G4Transportation(verboseLevel);
// Here we can adjust the parameters for this instance/particle, e.g.
transport->SetThresholdWarningEnergy(  30.0 * CLHEP::keV  );
transport->SetThresholdImportantEnergy( 3.0 * CLHEP::MeV );
transport->SetNumberOfTrials( 50 );

procManager->AddProcess(transport, -1, 0, 0);
// Add the new type of Transport(ation)
```

You can repeat this for positrons - but we recommend much lower thresholds for positrons as their annihilation will produce two 0.5 `MeV` gammas.

### 11.2.7 Avoiding loopers or reducing the incidence of looping particles

There are different ways to reduce the occurence of looping particles. This section will provide an overview, and refer the user to the detailed information on particle propagation in a magnetic field for details.

Volumes which have a strong field and contain vacuum, large air cavities or large volumes of gases are prime candidates for causing integration difficulties for low energy charged particles, which result in looping particles.

- A very simple way to reduce the incidence of looping particles is to reduce the maximum step size which particles that interact very infrequently can travel. Geant4 attempts to estimate an effective maximum using the diameter of the world volume, and frequently the maximum step size is large if the experimental hall is used as the world volume and has large dimentions. A smaller value can be impose by using the method. `G4PropagatorInField::SetMaximumStepSize()`
- Another ways is to change the maximum number of integration substeps. The default value is 1000, but it can be obtained from `G4PropagatorInField`

  ```
  auto *transportMgr = G4TransportationManager::GetTransportationManager();
  G4PropagatorInField* propFld= transportMgr->GetPropagatorInField();
  G4cout << " The maximum number of substeps for integration is "
         << propFld->GetMaxLoopCount() << G4endl;
  ```

  It can be also be changed simply by calling the corresponding set method, e.g.

  ```
  propFld->SetMaxLoopCount(2500);
  ```

- Assign a separate `G4FieldManager` class to each such volume. It can use adapted methods for the integration of the ODEs of motion.
- One solution is to use a helical stepper, such as `G4HelixImplicitEuler` or `G4HelixHeum` which are inherently for steps over multiple 'turns' of a helix-like track. Their reason d' etre is the ability to use the helix solution as baseline 'first-order' like solution and treat deviations from this as something like pertubations.
- A new integration driver `G4BFieldIntegrationDriver`, introduced in Geant4 10.6 samples the value of the magnetic field at the start of each step and using the estimated track curvature determines whether the current step will traverse an angler smaller or larger than $2 * \pi$. For larger steps the hybrid-helical stepper `G4HelixHeum` is used, and for smaller steps the `G4DormandPrince745` stepper is used. This driver is the default driver in Geant4 10.6, created by `G4ChordFinder` for magnetic fields, and when `G4FieldManager` 's `CreateChordFinder` method is called. Note that it is applicable only for charged particles in pure magnetic field.
- An older approach, usable before Geant4 10.6, is to use the `G4HelixMixedStepper`. This also combines a helix stepper for large steps with a Runge-Kutta stepper for small and intermediate step sizes. It does this by checking the value of the field at the start of every integration. As a result it is less efficient than the new method `G4BFieldIntegrationDriver`.

**Ensuring progress for particles**

To allow better progress for looping particles, the default behaviour was changed in Geant4 10.6, so that a track's parameters are changed within a Geant4 step after it has undertaken 100 integration substeps.

For the remainder of the current step the track 'tightness' parameter *delta chord* is relaxed, first by a factor of 2 at the hundredth step, and again by another factor of 2 after every 100 subsequent steps. The original value of *delta chord* is restored at the end of the current Geant4 step.

This will allow tracks which are in a tight spiral with a radius of curvature less than the user-defined *delta chord* to make substantial progress.

Note that this applies only to particles below the 'Important' energy threshold, which would be killed if their integration is not completed within a single Geant4 step.

**Status of this Document**

Guide for Application Developers using the GEANT4 toolkit.

- Rev 1.0: First Sphinx version implemented for GEANT4 Release 10.4, 8th Dec 2017
- Rev 2.0: Updates and fixes in documentatio for GEANT4 Release 10.4, 15th May 2018
- Rev 3.0: GEANT4 Release 10.5, 11th December 2018
- Rev 3.1: GEANT4 Updates and fixes - especially to search functionality, 5th March 2019
- Rev 4.0: GEANT4 Release 10.6, 6th December 2019
- Rev 5.0: GEANT4 Release 10.7, 4th December 2020
- Rev 6.0: GEANT4 Release 11.0, 10th December 2021
- Rev 7.0: GEANT4 Release 11.1, 9th December 2022
- Rev 7.1: GEANT4 Fixes to http links, 15th July 2023
- Rev 8.0: GEANT4 Release 11.2, 8th December 2023

# BIBLIOGRAPHY

[Booch1994] Grady Booch *Object-Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Co. Inc 1994 ISBN: 0-8053-5340-2

[Ellis1990] Margaret Ellis and Bjarne Stroustrup *Annotated C++ Reference Manual (ARM)*. Addison-Wesley Publishing Co. 1990

[Hecht1974] E. Hecht and A. Zajac *Optics*. Addison-Wesley Publishing Co. 1974 pp. 71-80 and pp. 244-246

[Janecek2010] M. Janecek, W. W. Moses, IEEE Trans. Nucl. Sci. 57 (3) (2010) 964-970 http://ieeexplore.ieee.org/document/5485130/

[Knoll1988] G.F. Knoll, T.F. Knoll and T.M. Henderson, Light Collection Scintillation Detector Composites for Neutron Detection, IEEE Trans. Nucl. Sci., 35 (1988) 872.

[Levin1996] A. Levin and C. Moisan, A More Physical Approach to Model the Surface Treatment of Scintillation Counters and its Implementation into DETECT, TRIUMF Preprint TRI-PP-96-64, Oct. 1996 https://inis.iaea.org/collection/NCLCollectionStore/_Public/29/030/29030591.pdf; https://doi.org/10.1109/NSSMIC.1996.591410

[Plauger1995] P.J. Plauger *The Draft Standard C++ Library*. Prentice Hall, Englewood Cliffs 1995

[RoncaliCherry2013] Roncali E & Cherry S 2013 *Simulation of light transport in scintillators based on 3D characterization of crystal surfaces.* (https://www.ncbi.nlm.nih.gov/pubmed/23475145) Phys. Med. Biol., Volume 58(7), p. 2185–2198.

[Roncali2017] Roncali et al. 2017 *An integrated model of scintillator-reflector properties for advanced simulations of optical transport.* (https://www.ncbi.nlm.nih.gov/pubmed/28398905) Phys. Med. Biol., Volume 62(12), p. 4811-4830.

[Stockhoff2017] Stockhoff et al. 2017 *Advanced optical simulation of scintillation detectors in GATE V8.0: first implementation of a reflectance model based on measured data.* (https://www.ncbi.nlm.nih.gov/pubmed/28452339) Phys. Med. Biol., Volume 62(12), L1-L8.