Geant4 Interface to Work with IAEA Phase-Space Files

Miguel A. Cortés-Giraldo[1], José M. Quesada[1], María I. Gallardo[1] and Roberto Capote[2]

[1] Dep. Física Atómica, Molecular y Nuclear, Universidad de Sevilla, Ap. 1065, E-41080 Sevilla, Spain
[2] NAPC–Nuclear Data Section, International Atomic Energy Agency, Vienna A-1400, Austria

December 14, 2009

Contents

1 Introduction to this guide

The aim of this document is to be the reference manual for a user that needs to read or write IAEA formatted phase-space files (PSF) in his/her Geant4 application. All the classes included in this interface are carefully explained in the following sections, therefore the user is encouraged to read through this manual before using them.

In section 2 a detailed description of *G4IAEAphspReader* and *G4IAEAphspWriter* classes is presented, including for each one a reference list with the available public methods. A list showing the IAEA public functions used in these classes is provided as well.

Section 3 explains how these files must be included in a Geant4 application. Next, the instructions of use, including some pieces of code as a brief examples, are given in order to provide the user a guide clarifying how to use these classes properly.

Finally, in section 4 some typical examples are shown. These are real examples that should work in a Geant4 application. Their goal is to show how the most important methods of these classes can be used.

2 Description of the interface

The complete Geant4 [1] interface for the use of phase-space files in IAEA format is composed of the following files:

- The IAEA routines published on its web site [2].
- The files defining the Geant4 reader class: `G4IAEAphspReader` (`.hh` and `.cc`)
- The files defining the Geant4 writer class: `G4IAEAphspWriter` (`.hh` and `.cc`)

The description of these files and classes is presented in the following sections.

2.1 The IAEA phase-space format

The IAEA phase-space (phsp) format is explained in the documentation published by the IAEA and available in the project web-site [2]. Therefore, and according to the scope of this manual, only the basics will be presented here. Interested reader is asked to review the original IAEA documentation provided on the cited reference.

As stated in [2], the IAEA phsp format was designed to cover both phase-space files and event generators. However, there are no event generators available yet, so we will deal with phase spaces in this contribution, This format is implemented in a set of read/write routines composed by the following files:

- `iaea_config.h`
- `iaea_header.h` and `iaea_header.cc`
- `iaea_phsp.h` and `iaea_phsp.cc`
- `iaea_record.h` and `iaea_record.cc`
- `utilities.h` and `utilities.cc`

IAEA function	Used in G4 classes?
`iaea_check_file_size_byte_order()`	Yes (R)
`iaea_copy_header()`	No
`iaea_destroy_source()`	Yes (R/W)
`iaea_get_constant_variable()`	Yes (R)
`iaea_get_extra_numbers()`	Yes (R)
`iaea_get_max_particles()`	Yes (R)
`iaea_get_maximum_energy()`	No
`iaea_get_particle()`	Yes (R)
`iaea_get_total_original_particles()`	Yes (R)
`iaea_get_type_extra_variables()`	Yes (R)
`iaea_get_used_original_particles()`	Yes (R)
`iaea_new_source()`	Yes (R/W)
`iaea_print_header()`	Yes (W)
`iaea_set_constant_variable()`	Yes (W)
`iaea_set_extra_numbers()`	Yes (W)
`iaea_set_paralell()`	No
`iaea_set_record()`	No
`iaea_set_total_original_particles()`	Yes (W)
`iaea_set_type_extrafloat_variable()`	No
`iaea_set_type_extralong_variable()`	Yes (W)
`iaea_update_header()`	Yes (W)
`iaea_write_particle()`	Yes (W)

Table 1: List of the public functions declared in the file *iaea_phsp.h*. More information about these functions can be found in the code itself or in the documentation provided in [2]. The right column indicates whether the function is invoked in the Geant4 interface or not. "R" means that it is used in *G4IAEAphspReader* class, whereas "W" means that it is called in *G4IAEAphspWriter* class.

Whenever the corresponding data of a certain phase space are recorded, these routines create two files with extensions `.IAEAphsp` and `.IAEAheader`, respectively. The `.IAEAphsp` file is a binary in which the previously asked data of all the particles passing through the defined plane is stored, including extra-variables when requested. Currently, the IAEA phsp format supports the storage of the following particles: photons, electrons, positrons, neutrons and protons. In addition, in the `.IAEAheader`, which is an ASCII file, the user can find information related to the phase-space data stored in the file, such as a statistical summary, number of original histories, number of particles of each kind that crossed the phase-space plane... and more.

2.1.1 Public functions of the IAEA routines

Table 1 shows a list of the public functions defined in the IAEA routines (left column). The descriptions of these functions are out of the scope of this manual, but more information can be found in the IAEA routines themselves, available on-line [2]. The user interested in this can have a look to file `iaea_phsp.h`, where a brief description of each function is included.

Most of these functions were implemented in the Geant4 interface. The right column of table 1 shows whether the function is used or not. If so, a "R" indicates that the IAEA

function is used in the reader class (*G4IAEAphspReader*), whereas a "W" means that the IAEA function is invoked in the writer class (*G4IAEAphspWriter*). Obviously 1, the functions that create and destroy an IAEA phsp object are used in both Geant4 classes. Not all functions of the IAEA interface have been used so far. There are utilities not needed for the current version, but they might be included in future versions of the release. If the reader needs to use any of them, he/she should contact authors, and an updated version would be prepared and released.

2.2 G4IAEAphspReader class

2.2.1 Description

This class works as another primary particle generator in Geant4, like *G4ParticleGun* or *G4GeneralParticleSource*, which are the most used by Geant4 users. Like these two classes, *G4IAEAphspReader* is derived from *G4VPrimaryGenerator* virtual class. Therefore, this new class should be used in a Geant4 application in the same way as the commonly used generators.

G4IAEAphspReader class is coded with the goal of providing the user a new tool for obtaining all the relevant information stored in a IAEA phase-space file. This class respects the internal structure of the IAEA routines, so that only the public functions listed in table 1 are used. With this policy, this Geant4 class will not be affected by futures internal (private) changes in the IAEA format implementation as long as the declared IAEA public functions and parameters remain frozen.

Among all the utilities that *G4IAEAphspReader* class has, the main ones are the following:

- In order to do a proper statistical analysis, it is very important to give the user the possibility of taking into account existing correlations. The class has been designed so that all generated particles that may have correlations between them are generated in the same Geant4 event. Correlations mostly come from two different sources: (a) particles sharing the same *original history* and (b) particles *recycled* several times. The user can set the number of times that each particle is recycled by means of the method *SetTimesRecycled(int)*. The criterium followed for the argument is the same as the one used in the code EGSnrc [3, 4], therefore '0' means that each particle is used only once, and n means that each particle is used $n + 1$ times.

  In Geant4, the most direct way to keep all the possible correlations between particles is by generating all these correlated particles in the same Geant4 event. This part is very important for the user because it means that one Geant4 event represents one (and only one) *independent event* and it doesn't corresponds to one unique particle. Therefore, all the particles stored in the PSF that share the same original history are thrown in the same event. In addition, in case of recycling, each particle would be repeated the desired amount of times, always in the same Geant4 event. The most important fact that comes out of this paragraph is that the user must bear in mind that *a certain number of events in a Geant4 run corresponds to that number of statistically independent events, and not to that number of single particles.* In other words, in one Geant4 event all the particles coming from the same original history are thrown, and they are repeated as many times as the user has set previously.

- With the methods *SetTotalParallelRuns(int)* the PSF can be divided into fragments. Therefore, the user can prepare parallel runs in different machines (CPU's) by using only a fixed fragment in each machine. The method *SetParallelRun(int)* is coded to choose one of these fragments.

- This interface allows the user to make any translation or rotation of the phase-space plane. Rotations can be performed around any axis of the global reference frame, in arbitrary order. Moreover, the user can define the direction of the rotation axis for both the gantry and the treatment head of the medical linac, and rotate the phase-space plane around each of these axes.

2.2.2 Reference guide

The public methods of *G4IAEAphspReader* class are listed below in groups:

- Method inherited from *G4VPrimaryGenerator* class which implementation is mandatory:
  - `void GeneratePrimaryVertex(G4Event* )`: it must be called from the User Primary Generator Action class that has been defined by the user in his/her Geant4 application. This Primary Generator Action must be derived from *G4VUserPrimaryGeneratorAction* virtual class.
- Methods needed to configurate "parallel" runs and recycling:
  - `void SetTotalParallelRuns(G4int n)`: the PSF will be divided into n fragments when the Geant4 run starts. The argument must be an integer greater than or equal to 1.
  - `void SetParallelRun(G4int m)`: the Geant4 run will use the particles stored in the m-th part of the PSF. The argument must satisfy the condition $1 \leq m \leq n$.
  - `void SetTimesRecycled(G4int r)`: each particle will be used r more times (EGSnrc criterium), so if $r = 0$ each particle is used once, if $r = 1$ they are used twice and so on.
- Methods needed to make spatial transformation to the position and momenta of the particles. The transformation that is applied first is the translation of the phase-space plane. This is followed by the rotations around the axis of the global coordinate system. At the end, the rotations around the isocenter (if defined) are considered. Two different rotations can be applied around the isocenter. If defined, the first one that it is applied is the rotation around the treatment head axis. Later, the rotation around the gantry rotation axis is considered.
  - `void SetGlobalPhspTranslation(const G4ThreeVector& )`: it changes the initial position of the particles from their original places stored in the PSF.
  - `void SetRotationOrder(G4int )`: a 3-digit integer, consisting of the 6 permutations of '1', '2' and '3', must be passed as argument. '1' refers to X global axis, '2' to Y global axis and '3' to Z global axis. The order in which the number appear sets the rotation sequence around the axis of the global frame. So, for example: '213' means that rotation sequence is first around Y axis, later around X axis and finally around Z axis.
  - `void SetRotationX(G4double )`: it sets the rotation angle around the X axis of the global frame.
  - `void SetRotationY(G4double )`: it sets the rotation angle around the Y axis of the global frame.

- `void SetRotationZ(G4double )`: it sets the rotation angle around the Z axis of the global frame.
- `void SetIsocenterPosition(const G4ThreeVector& )`: it sets the position of the isocenter related to the global coordinate system. This method is mandatory when the user applies isocentric rotations to the phase-space file by calling the methods defined next.
- `void SetCollimatorRotationAxis(const G4ThreeVector& )`: in a medical linac, this sets the direction of the rotation axis of the treatment head.
- `void SetGantryRotationAxis(const G4ThreeVector& )`: in a medical linac, this sets the direction of the rotation axis of the gantry.
- `void SetCollimatorAngle(G4double )`: it sets the rotation angle for the treatment head.
- `void SetGantryAngle(G4double )`: it sets the rotation angle for the gantry.

- Methods for getting information about the phase-space file:
  - `G4String GetFileName()`: it returns the file name (without extension).
  - `G4int GetSourceReadId()`: it returns the source ID (integer number) which is used internally in the IAEA routines to identify this IAEA phsp file.
  - `G4long GetOriginalHistories()`: it returns the number of original histories stored in the PSF.
  - `G4long GetUsedOriginalParticles()`: it returns the number of original histories already used in the PSF.
  - `G4long GetTotalParticles()`: it returns the total number of particles stored in the PSF.
  - `G4long GetTotalParticlesOfType(G4String )`: it returns the number of particles of one kind stored in the PSF. The possibilites are: `"ALL"`, `"PHOTON"`, `"ELECTRON"`, `"POSITRON"`, `"NEUTRON"` and `"PROTON"`.
  - `G4int GetNumberOfExtraFloats()`: it gives the number of extra float variables considered in the PSF.
  - `G4int GetNumberOfExtraInts()`: it gives the number of extra integer variables considered in the PSF.
  - `std::vector<G4int>* GetExtraFloatsTypes()`: a vector is returned with the numbers that indicates the type of each extra float variable.
  - `std::vector<G4int>* GetExtraIntsTypes()`: it returns a vector which stores the numbers that represent the type of each extra integer variable.
  - `G4double GetConstantVariable(const G4int index)`: it returns the value of the variable represented by *index* if declared constant. The correspondence between index and variable is: $0 = x$, $1 = y$, $2 = z$, $3 = u$, $4 = v$, $5 = w$ and $6 = wt$.

- Methods that return the particle-related variables stored in a event. All of them return a vector that stores the variables asked. One given particle is referred by the same component in all the vectors, provided they're called during the same event.

  - `std::vector<G4int>* GetParticleTypeVector()`: it returns a vector containing the integers that identify the particles passing through the phase-space plane in the given event.
  - `std::vector<G4double>* GetEnergyVector()`: it returns a vector containing the kinetic energy of each particle passing through the phase-space plane in the given event.
  - `std::vector<G4ThreeVector>* GetPositionVector()`: it returns a vector which contains the position of the particle when crossing the phase-space plane.
  - `std::vector<G4ThreeVector>* GetMomentumVector()`: it returns a vector containing the direction cosines of the linear momentum for each particle when it's crossing the phase-space plane.
  - `std::vector<G4double>* GetWeightVector()`: it returns a vector which stores the statistical weight for each particle passing through the phase-space plane.
  - `std::vector< std::vector<G4double> >* GetExtraFloatsVector()`: it is used to obtain the vector which stores the float-type extra variables for each particle.
  - `std::vector< std::vector<G4long> >* GetExtraIntsVector()`: it is used to obtain the vector which stores the integer-type extra variables for each particle.

- Methods which return information about number of fragments which the PSF is divided into and about recycling:

  - `G4int GetTotalParallelRuns()`: it returns the number of partitions considered for the PSF.
  - `G4int GetParallelRun()`: it returns which partition is goint to be used during the Geant4 run.
  - `G4int GetTimesRecycled()`: it returns the number of times which each particle is going to be recycled during the Geant4 run.

- Methods coded to return all the information on the spatial transformations that will be made to the position and momentum of each particle stored in the phase-space file before generating them in the Geant4 simulation.

  - `G4ThreeVector GetGlobalPhspTranslation()`: it returns the displacement considered for the phase-space plane.
  - `G4int GetRotationOrder()`: it returns the order in which the rotations around the global frame axis are applied.
  - `G4double GetRotationX()`: it gives the rotation angle around the X axis of the global coordinate system.
  - `G4double GetRotationY()`: it gives the rotation angle around the Y axis of the global coordinate system.
  - `G4double GetRotationZ()`: it gives the rotation angle around the Z axis of the global coordinate system.
  - `G4ThreeVector GetIsocenterPosition()`: it returns the position of the isocenter, defined in the global coordinate system.
  - `G4ThreeVector GetCollimatorRotationAxis()`: it returns the direction of the rotation axis of the treatment head for the machine used to generate this PSF.

- `G4ThreeVector GetGantryRotationAxis()`: it returns the direction of the rotation axis of the gantry for the machine used to generate this PSF.
- `G4double GetCollimatorAngle()`: it returns the rotation angle around the rotation axis of the collimator for the machine used to generate this PSF.
- `G4double GetGantryAngle()`: it returns the rotation angle around the rotation axis of the gantry for the machine used to generate this PSF.

2.3 G4IAEAphspWriter class

The writing of one or several IAEA-formatted PSFs is managed by this class. The design of this class is different from the design of the reader class in the sense that, for the writing task, only one *G4IAEAphspWriter* object is needed. Therefore, *G4IAEAphspWriter* is a singleton class with takes over the storage of the information in the IAEA phsp files during the run. The design of the reader class is different, and for that case one object is needed for each IAEA phsp file used during the Geant4 run.

As explained in the IAEA documentation [2], the phase space is written in a binary file which extension is `.IAEAphsp`. It stores information about energy E, statistical weight w, the three components of the position (x, y, z), the direction cosines (u, v, w) and the extra variables either integer (*extraints*) or floating point reals (*extrafloats*). To save space in disk, the user can declare as constant any coordinate, direction cosine and/or the particle weight. The PSF generated by means of this class defines z as constant, and store the *incremental history number* (called *n_stat* in Penelope) as a integer extra variable. In total, 33 bytes are allocated for each particle in the file. All this information is contained in the *.IAEAheader* file.

After a Geant4 run, two IAEA phsp files are generated (a header and binary files). The default name would be `"PSF"` followed by underscore and the value of coordinate z for that scoring plane with corresponding extensions. A Geant4 application may include several runs for the same execution. In these cases, the corresponding Geant4 run ID is included at the end of the name.

This class also book-keeps the particles that cross any scoring plane during a event. If the same particle crossed it again, it wouldn't be counted once more, avoiding multiple passers.

2.3.1 Reference guide

The public methods of *G4IAEAphspReader* class are listed below in groups:

- Static method to get the singleton reference:
  - `static G4IAEAphspWriter* GetInstance()`
- Methods that must be called in the User Actions (UAs) of the Geant4 application. These methods take over the process of writing a IAEA phsp file. In section 3 can be found how to use them properly:
  - `void BeginOfRunAction(const G4Run* )`: this method basically creates the PSF's just before the Geant4 run starts.
  - `void EndOfRunAction(const G4Run* )`: this method is key to save the PSF's at the end of the Geant4 run.
  - `void BeginOfEventAction(const G4Run* )`: in this methods some counters are reset.

  - `void UserSteppingAction(const G4Run* )`: this method checks whether the particle has crossed any of the scoring planes.

- "Set" and "Get" public methods:

  - `void SetZStop(G4double zValue)`: it defines a new scoring plane at $z =$ `zValue`. This method must be called **at least once** before the Geant4 run starts.
  - `void SetFileName(G4String )`: it sets the name of the IAEA phsp file (without extension). `"PSF"` is the name by default.
  - `G4String GetFileName()`: it returns the name of the IAEA phsp file (without extension).
  - `std::vector<G4double>* GetZStopVector()`: it returns a vector containing the position (z coordinate) of the defined scoring planes.

- Public methods to be called by the user at his/her convenience:

  - `void UpdateHeaders()`: it updates all the *IAEAheader* files created during the run. The user is suggested to invoked it in the "Event Action" class of the Geant4 application.

3 How to use this interface in a Geant4 application

3.1 First step

1. The files listed in section 2 must be copied in the directory of the Geant4 application. As usual, header files (`.h` or `.hh`) should be copied into the *include* subdirectory, whereas source files (`.cc` or `.cpp`) should be copied into *src* subdirectory.

2. The `.cpp` files must be renamed as `.cc` files. Otherwise they would not be properly compiled by *GNUmakefile*.

3. After placing and renaming the files as indicated, the Geant4 application must be compiled. No errors should appear.

4. The next step is to prepare the application to read and/or write IAEA phsp files. This includes changes in the Geant4 application source code, which are explained in the following paragraphs. These changes may require the user to create new User Actions in the Geant4 application.

5. Once the suitable changes has been made, a compilation is mandatory. If these changes are done properly, there should not be compilation errors.

3.2 How to read a IAEA phsp file

In order to use a IAEA phsp as a source of particles in a Geant4 application, the user must fulfill the following requeriments in the Primary Generator class of this application:

1. Add a pointer to *G4IAEAphspReader* object in the header file as a data member of the Primary Generator class.

2. In the Primary Generator class constructor: the pointer to *G4IAEAphspReader* object must be created, passing the name of the phase-space file as argument (without including the IAEA extension). It is very important to remark that, up to date, the reader class can deal with only one IAEA phsp file. In next versions, it is possible that the reader class may be able to manage more than one IAEA phsp file as a source.

3. In the *GeneratePrimaries()* method of the Primary Generator class: the pointer to *G4IAEAphspReader* object must be used to invoke the *GeneratePrimaryVertex()* method of this pointer.

These minimum requirements are shown in the following example, where the Primary Generator class is called *PrimaryGeneratorAction*.

Example 1

Header file:

```
#ifndef PrimaryGeneratorAction_h
#define PrimaryGeneratorAction_h 1

#include "G4VUserPrimaryGeneratorAction.hh"

class G4Event;
class G4IAEAphspReader;

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

  void GeneratePrimaries(G4Event*);

private:
  // Phase space reader
  G4IAEAphspReader* theIAEAReader;
};

#endif
```

Source file:

```
#include "PrimaryGeneratorAction.hh"
#include "G4Event.hh"
#include "G4IAEAphspReader.hh"

PrimaryGeneratorAction::PrimaryGeneratorAction()
{
  G4String fileName = "PSF";
  theIAEAReader = new G4IAEAphspReader(fileName);
}

PrimaryGeneratorAction::~PrimaryGeneratorAction()
```

```
{
  if (theIAEAReader) delete theIAEAReader;
}

void PrimaryGeneratorAction::GeneratePrimaries(G4Event* anEvent)
{
  theIAEAReader->GeneratePrimaryVertex(anEvent);
}
```

In case that some of the utility methods (for partitioning or spatial transformation) are needed by the user, they can be invoked in the same way as the modifier methods of the Primary Generator classes commonly used (*G4ParticleGun* or *G4GeneralParticleSource*). In section 4 some examples on these methods are provided.

3.3 How to write IAEA phsp files

To make the Geant4 application capable to write IAEA phsp files, the user must fulfill these minimum requirements:

1. This interface needs the Geant4 application to have these optional User Actions defined:
   - "Run Action" class (derived from *G4UserRunAction*).
   - "Event Action" class (derived from *G4UserEventAction*).
   - "Stepping Action" class (derived from *G4UserSteppingAction*).
2. In the "Run Action" class, these lines must be included:
   - In *BeginOfRunAction()* method: the reference of the singleton must be obtained in order to invoke the *SetZStop()* method of the singleton. This method should be called once for each scoring plane, passing the z value in the argument. Once all the z value has been stored in the previous step, the particular method of the singleton called *BeginOfRunAction()* must be used in the following line.
   - In *EndOfRunAction()* method: the user must access to the singleton by means of *G4IAEAphspWriter::GetInstance()*, if needed, and invoke the particular *EndOfRunAction()* method of the singleton.

   Let *RunAction* be the name of the "Run Action" class. The following lines should appear in the source file:

```
#include "RunAction.hh"

#include "G4Run.hh"
#include "globals.hh"
#include "G4IAEAphspWriter.hh"


RunAction::RunAction()
{}

RunAction::~RunAction()
```

```
{}

//-----
void RunAction::BeginOfRunAction(const G4Run* aRun)
{
  G4IAEAphspWriter* IAEAWriter = G4IAEAphspWriter::GetInstance();
  IAEAWriter->SetZStop(10.*cm);
  // Add more ZStops if needed:
  //  IAEAWriter->SetZStop(50.*cm);
  //  IAEAWriter->SetZStop(90.*cm);
  IAEAWriter->BeginOfRunAction(aRun);
}

//-----
void RunAction::EndOfRunAction(const G4Run* aRun)
{
  G4IAEAphspWriter::GetInstance()->EndOfRunAction(aRun);
}
```

3. In the "Event Action" class: the user only needs to include one line in the *BeginOfEventAction()* method. In this line a pointer to the singleton must be obtained, and the *BeginOfEventAction()* method of this singleton must be invoked.

   The source file of a "Event Action" with this minimum requirement should be like the following example:

```
#include "EventAction.hh"

#include "G4Event.hh"
#include "G4IAEAphspWriter.hh"


EventAction::EventAction()
{}

EventAction::~EventAction()
{}

//-----
void EventAction::BeginOfEventAction(const G4Event* aEvent)
{
  G4IAEAphspWriter::GetInstance()->BeginOfEventAction(aEvent);
}
```

4. In the "Stepping Action" class: only one line is needed here. The user must use the singleton method called *UserSteppingAction()*, as indicated in the following example:

```
#include "SteppingAction.hh"

#include "G4Step.hh"
#include "G4IAEAphspWriter.hh"
```

```
SteppingAction::SteppingAction()
{}

SteppingAction::~SteppingAction()
{}

//-----
void SteppingAction::UserSteppingAction(const G4Step* aStep)
{
  G4IAEAphspWriter::GetInstance()->UserSteppingAction(aStep);
}
```

4 Examples

Example 2: source file of a Primary Generator class which generates the particles from a IAEA phsp file called `PSF_afterJaws.IAEAphsp`. Its corresponding gantry has been rotated 45 degrees, and the treatment head has been rotated 90 degrees. The isocenter is placed at $\mathbf{r}_{\text{isoc}} = (0, 0, 1)$ m. In addition, this PSF will be divided into 8 fragments in order to create 8 parallel jobs, and the particles are taken from the 3^{rd} fragment in this particular job. Each particle will be recycled 24 times, so they will be used 25 times.

Source file:

```
#include "PrimaryGeneratorAction.hh"
#include "G4Event.hh"
#include "G4IAEAphspReader.hh"

PrimaryGeneratorAction::PrimaryGeneratorAction()
{
  G4String fileName = "PSF_afterJaws";
  theIAEAReader = new G4IAEAphspReader(fileName);

  // define the position of the isocenter
  // it's mandatory before rotations around this point
  G4ThreeVector isoPos(0., 0., 100.*cm);
  theIAEAReader->SetIsocenterPosition(isoPos);

  // Define the rotations
  // (The treatment head rotation is applied first!)
  G4ThreeVector collimAxis(0., 0., 1.);
  G4ThreeVector gantryAxis(0., 1., 0.);
  theIAEAReader->SetCollimatorAxis(collimAxis);
  theIAEAReader->SetCollimatorAngle(90.*deg);
  theIAEAReader->SetGantryAxis(gantryAxis);
  theIAEAReader->SetCollimatorAngle(45.*deg);

  // Now define the partition to be used, how many they are
  // and recycling
  theIAEAReader->SetTotalParallelRuns(8); // 8 fragments
  theIAEAReader->SetParallelRun(3);       // 3rd fragment of the PSF
  theIAEAReader->SetTimesRecycled(24);    // particles used 25 times
```

```
}

PrimaryGeneratorAction::~PrimaryGeneratorAction()
{
  if (theIAEAReader) delete theIAEAReader;
}

void PrimaryGeneratorAction::GeneratePrimaries(G4Event* anEvent)
{
  theIAEAReader->GeneratePrimaryVertex(anEvent);
}
```

Example 3: source file of a Primary Generator class which generates the particles from a IAEA phsp file called `PSF_beforeJaws.IAEAphsp`. They were generated considering the isocenter at $(0, 0, 1)$ m, but the simulation considers the isocenter at the origin of the coordinate reference system. Two rotations must be done: first around the Z axis, 90 degrees, and later around Y axis, 45 degrees. In other words, the same rotations as in the previous example.

Source file:

```
#include "PrimaryGeneratorAction.hh"
#include "G4Event.hh"
#include "G4IAEAphspReader.hh"

PrimaryGeneratorAction::PrimaryGeneratorAction()
{
  G4String fileName = "PSF_beforeJaws";
  theIAEAReader = new G4IAEAphspReader(fileName);

  // RECOMENDATION: call the spatial transformations in the
  // order in which they are applied (see user manual)

  // phase-space plane shift
  G4ThreeVector psfShift(0., 0., -100.*cm);
  theIAEAReader->SetGlobalPhspTranslation(psfShift);

  // rotations
  theIAEAReader->SetRotationOrder(321);
  theIAEAReader->SetRotationZ(90.*deg);
  theIAEAReader->SetRotationY(45.*deg);
}

PrimaryGeneratorAction::~PrimaryGeneratorAction()
{
  if (theIAEAReader) delete theIAEAReader;
}

void PrimaryGeneratorAction::GeneratePrimaries(G4Event* anEvent)
{
  theIAEAReader->GeneratePrimaryVertex(anEvent);
}
```

References

[1] S. Agostinelli, J. Allison, K. Amako, J. Apostolakis, *et al*, "Geant4 – a simulation toolkit ", Nucl. Instr.& Meth. Phys. Res. **A506** (2003) 250-303.

[2] R. Capote and I. Kawrakow, "Read/write routines implementing the IAEA phsp format", version of December 2009. Available online at *http://www-nds.iaea.org/phsp/software/iaea_phsp_Dec2009.zip#phsp_rw*

[3] I. Kawrakow, "Accurate condensed history Monte Carlo simulation of electron transport. I. EGSnrc, the new EGS4 version" , Med. Phys. **27** (2000) 485–498.

[4] I. Kawrakow and D. W. O. Rogers, "The EGSnrc code system: Monte Carlo simulation of electron and photon transport" , Tech. Rep. **PIRS-701**, National Research Council of Canada, Ottawa, Canada, 2000.

[5] F. Salvat, J. M. Fernández-Varea, E. Acosta and J. Sempau, "PENELOPE–A Code System for Monte Carlo Simulation of Electron and Photon Transport", Issy-les-Moulineaux: OECD Nuclear Energy Agency, 2001.