# PenRed: An extensible and parallel Monte-Carlo simulation framework for radiation transport based on PENELOPE

**V. Giménez-Alventosa[1], V. Giménez Gómez[2], and S. Oliver[3]**

[1]Instituto de Instrumentación para Imagen Molecular (I3M), Centro mixto CSIC - Universitat Politècnica de València , Camí de Vera s/n, 46022, València, Spain, vicent.gimenez@i3m.upv.es
[2]Departament de Física Teòrica and IFIC, Universitat de València-CSIC , Dr. Moliner, 50, 46100, Burjassot, València, Spain, vicente.gimenez@uv.es
[3]Instituto de Seguridad Industrial, Radiofísica y Medioambiental (ISIRYM), Universitat Politècnica de València, Camí de Vera s/n, 46022, València, Spain, sanolgi@upvnet.upv.es

February 27, 2020

**Abstract**

The present document is the manual of the PenRed code system. PenRed is a general-purpose, stand-alone and extensible framework code based on PENELOPE for parallel Monte Carlo simulations of radiation transport through matter. It has been implemented in C++ programming language and takes advantage of modern object-oriented technologies. In addition, PenRed offers the capability to read and process DICOM images on which it can construct and simulate voxelized geometries so as to facilitate its usage in medical applications. Our framework has been successfully tested against the original PENELOPE Fortran code.

## 1 Introduction

The present document describes the basic concepts of PenRed code structure and contains all the necessary information to create custom modules or extensions of the code.

PenRed is a fully parallel, modular and customizable framework for Monte Carlo simulations of the passage of radiation through matter. It is based on the PENELOPE [1] code system, from which inherits its unique physics models and tracking algorithms for charged particles. For further details on the interactions models, the reader is referred to the excellent PENELOPE manual [2]. PenRed has been coded in C++ following an object-oriented programming paradigm restricted to the C++11 standard. Our engine implements parallelism via a double approach: on the one hand, by using standard C++ threads for shared memory, improving the access and usage of the memory, and, on the other hand, via the MPI standard for distributed memory infrastructures. Notice that both kinds of parallelism can be combined together in the same simulation. In addition, PenRed provides a modular structure with methods designed to easily extend its functionality. Thus, users can create their own independent modules to adapt our engine to their needs without changing the existing modules. Furthermore, user extensions will take advantage of the built-in parallelism without any extra effort or knowledge of parallel programming.

The distribution package consists of all the source code organized in different folders as follows. The `geometry` folder contains the classes for performing the tracking of particles through material systems in quadric and voxelized geometries based on the PENGEOM package of PENELOPE, PENCT program and PENEASY. The folder `particleGen` includes the random generators for initial particle states, which description and usage can be found at section 6. The `kernel` folder contains the components which form the core of PenRed code system. They are organized in subfolders and describe in section 4.1. Some both auxliary and basic libraries are included in the folder `lib`. The folder `tallies` contains the tallies currently implemented in PenRed to extract information from the Monte Carlo simulations, as deposited energy, fluence, etc. Some utilities can be found in the folder `utilities`, such us a program to convert from binary to ASCII format PenRed phase space files, a geometry voxelizer etc. Also, the folder `tests` contains some tests used to check the correct operation of the auxiliary libraries. Finally, a generic main program `pen_main` is provided in the folder `mains`. It performs simulations for radiation (electron, positron and photon) sources of various kinds and material systems with quadric and voxelized geometries. With `pen_main`, several variance reduction techniques can be employed as interaction forcing and particle splitting (section 6.6).

The manual is organized as follows. In section 2 shows a useful guide of installation and execution of the framework to run simulations. Section 3 provides some information of the internal data structures format. Then, section 4 describes the engine components of PenRed discussing its implementation details of the kernel, geometries, string instantiator mechanism, state samplers and tallies. Take into account that, for a complete understanding of this section some knowledge about classes, inheritance, template functions and classes as well as other C++ features of 2011 standard are required. In section 5 a briefly description of important definitions, constants, parameters and units is shown. Finally, section 6, shows how to execute the main program provided with PenRed package allowing the user to perform simulations with examples of all the tallies implemented, sources and geometries.

For users only interested in package usage, go to sections 3 and 6, which explain the internal data used in configuration files and how to use the tallies, sources, geometries and other simulation parameters respectively.

In addition, section 5 summarise some useful definitions necessary to understand the code and its operation.

# Contents

## 2 Installation and execution

PenRed depends on few external libraries which should be installed to activate the corresponding features (see below). It is the user's responsibility to check that these external dependencies are installed before activating a feature. The user should refer to the appropriate installation guides of the optional packages.

The recommended configuration for compiling PenRed is the following:

- g++ of GCC 4.7.3 to 9.2 with support for C++ 11 standard. However, PenRed can also be compiled with Intel icc version 2019 and clang version 8.
- CMake minimal version 3.4.

The installation of PenRed is very easy. First, download PenRed sources from our GitHub repository,

```
git clone https://github.com/PenRed/PenRed.git
```

The code must be compiled in the `src` folder. Thus, move into this folder. To simplify the installation, PenRed includes a CMake file and a bash script (`compile.sh`) to compile the code automatically. In this script, you can enable/disable the following optional features,

- DICOMs: If it is enabled, PenRed capabilities to read and simulate DICOM images will be active. This option requires the library dicom toolkit (dcmtk) [3].
- Multi-threading: This option enables multi-threading capabilities. PenRed implements multi-threading via the standard thread library specified in the C++11 standard. Thus, it is not required any extra library to enable this option.
- MPI: This option enables MPI simulations. It requires a library with an implementation of the MPI standard, such as openmpi [4] or mpich [5].

Notice that all previous dependencies are optional. The corresponding libraries can be found at most linux package repositories. For example, to compile PenRed with DICOM support in Fedora, you can use the `dnf` command to install the dependencies,

```
sudo dnf install dcmtk dcmtk-devel
```

After that simple step, you can launch the compilation using the provided script,

```
bash compile.sh
```

or doing it yourself,

```
cd /path/to/PenRed/repository/src

mkdir build

cd build

ccmake ../
```

```
make -jN (with N the number of processes)

make install
```

With `ccmake` you can configure the optional PenRed features with a more friendly interface. But, of course, you can use directly `cmake ../` defining the appropriate flags like,

```
   cmake -DWITH_DICOM="ON" -DWITH_MULTI_THREADING="ON"
-DWITH_MPI="OFF" -DDEVELOPMENT_WARNINGS="OFF" ../
```

Once the code has been compiled, the user can found the executable of our provided main program ready to simulate at,

*src/compiled/mains/pen_main*

To execute the program, the user needs a configuration file and, probably, the required data base files, such as material and geometry files. Their path should be specified at the configuration, remaining the configuration file path as the only program argument. So, to execute the program, the user must use

*./pen_main path/to/configuration/file*

Otherwise, if the MPI capabilities have been enabled at the compilation, the code should be executed as any MPI program, for example,

*mpirun -np Nprocesses ./pen_main path/to/configuration/file*

where *Nprocesses* specify the number of MPI processes to use. Of course, the user can use any other options of the *mpirun* command, such us specify the hosts where execute the code via the *hostfile* option.

## 3 Internal data format

### src/kernel/parsers

To provide a unified format for input and configuration, PenRed implements a set of data structures. This implementation can be found at the following files,

*src/kernel/parsers/includes/pen_data.hh*

and

*src/kernel/parsers/source/pen_data.cpp*

The format used in PenRed is basically a *key/value* pair where the key format is based on unix folder system, i.e. a generic key is a string with the following structure:

$/folder1/folder2/.../element$

On the other hand, the value can be a number, bool (True or False), character, string or an array of numbers, bools or characters. Notice that these types can be combined in a single array. Also, strings must be made with double cuotes ("text") to be parsed correctly. For example, code 1 shows a configuration for a cylindrical dose distribution tally.

```
tallies/cylDoseDistrib/type "CYLINDRICAL_DOSE_DISTRIB"
tallies/cylDoseDistrib/print-xyz true
tallies/cylDoseDistrib/rmin 0.0
tallies/cylDoseDistrib/rmax 30.0
tallies/cylDoseDistrib/nbinsr 60
tallies/cylDoseDistrib/zmin 0
tallies/cylDoseDistrib/zmax 30.0
tallies/cylDoseDistrib/nbinsz 60
```

Code 1: Internal data example

### 3.1 Implementation

This section provides a brief explanation of internal data implementation. PenRed internal data structure consists on four classes where each one uses the previous class to create a more complex structure. First, *pen_parserData* is basically a union with four allowed types: *char, int, double* and *bool*. Notice that *pen_parserData* can't be a string or pointer. Next, *pen_parserArray* is a standard C++ vector of *pen_parserData* variables. The third class is *pen_parserElement*, which can store a C++ string or a *pen_parserArray* variable. Finally, *pen_parserSection* stores a C++ map where each key-value pair uses the types,

$< std::string|pen\_parserArray >$

where the left element is the pair key and the right element is the pair value. As we mentioned at previous section, key format follows the structure of Unix paths, i.e. ,

*/folder1/folder2/element.*

In our schema, each "folder" will be considered as a "section". On the other hand, a key with no sub-folders is considered an "element". Notice that keys whose include an element path as a section are not allowed and the program will return an error or overwrite this element according to the called function. Thus, the following configuration can't exist in a input format map structure,

*/folder1/folder2/element*
*/folder1/folder2/element/element2*

since *element* is not a section and therefore cannot contain any other section nor element.

For further information and usage examples, consult the provided examples in *src/tests/internalData* folder.

## 4 Engine components

This section shows the internal components of PenRed engine. For users interested only on the package usage, go to section 6. Notice that this section discus implementation details of our engine, thus the user requires some knowledge about classes, inheritance, template functions and classes, and other C++ features present in the 2011 standard. However, this is not necessary to understand how to use the framework (section 6).

## 4.1 Kernel Modules

**src/kernel**

In this section we will introduce the PenRed kernel, which is made up of the following components,

- Materials
- Contexts
- Particles
- Interactions
- Particle states
- Grids
- Random generator

Most of the components listed above have been implemented as an abstract template classes in order to achieve a good balance between performance and flexibility. At the following sections we will describe the characteristics of each component and their relations and dependencies.

### 4.1.1 Random generator

**src/kernel/rands**

Actually, PenRed uses the same random generator as PENELOPE FORTRAN version. This is an adapted version of subroutine RANECU written by F. James [6], which has been modified to give a single random number at each call. However, the random generator has been implemented into a class to allow to create independent random generators instances with their corresponding private seeds to support multi-threading simulations. All methods and functions that require random numbers, such as *knock*, state samplers, etc., requires one of these classes as argument.

### 4.1.2 Particle states

**src/kernel/states**

Particle states are simple structures that store the state of a single particle. The state of a particle means the required variables to describe the physical characteristics of the particle and its position in the geometry system. PenRed provides a base state class called *pen_particleState*, which is compatible with all other components. If the user needs to extend the particle state to store extra particle characteristics, it can be achieved creating a subclass of the *pen_particleState* class. We strongly recommend following this procedure to ensure compatibility with other PenRed components.

Basically a generic state contains the following variables,

- **E**: Particle energy in eV.
- **X,Y,Z**: Particle position in cm.
- **U,V,W**: Normalized direction vector.

- **PAGE**: Particle age in seconds.
- **ILB**: Array with 5 components to store particle metadata information, such as parent particle type (see table 1).
- **IBODY**: Geometry body index where the particle is located.
- **MAT**: Geometry material index where the particle is located.
- **WGHT**: Particle weight which can be modified by variance reduction techniques.
- **LAGE**: Boolean to enable/disable time of flight calculations.

An example of an extended particle state can be found in *src/kernel/gammaPol_state.hh* file, where we provide a state with extra variables to store the Stokes parameters to characterize polarized photons.

#### 4.1.3 State Stacks

**src/kernel/includes/pen_classes.hh**

State stacks are simply particle state buffers defined in *src/kernel/includes/pen_classes.hh* file. These components should be used to store the states of secondary particles generated during the simulation.

Since the state stacks have been implemented as template classes, they can be used for any user defined particle state. However, notice that the *store* and *getState* methods use the operator "=". So, if a user defined state contains dynamic memory allocations, references or pointers to non static member data, she/he must care about overloading operator "=" to ensure that the copy operation is handled correctly.

#### 4.1.4 Grids

**src/kernel/grids**

The Grid abstract class *abc_grid*, declared in *src/kernel/includes/pen_classes.hh*, defines a pattern to implement different kinds of grids. These grids can be used, for example, to sample interaction cross sections for different values of the incident energy, to store simulation information etc.

The common grid member variables are listed and explained below. Notice that the specifier *Raw* used in the definition of some variables means that no transformation has been applied to their values. For example, in a logarithmic scale, if the lowest grid value is log(5), the lowest *raw* grid value is 5.

- **EMIN**: Minimum grid value.
- **EL**: Raw lowest grid value, typically $0.99999 \cdot EMIN$ to avoid rounding problems.
- **EU**: Raw highest grid value.
- **ET**: Array of raw values of each grid bin.
- **DLEMP**: Array that stores transformed values of ET. For example, in a logarithmic scale, the element with index $i$ of DLEMP must be calculated as $\log(ET[i])$, following C/C++ syntax.

- **DLFC**: Inverse of distance between transformed bins. We suppose that the transformed grid bins are evenly spaced.
- **DLEMP1**: Transformed value of EL. For example, in a logarithmic scale, its value must be $\log(EL)$.

The grid base class *abc_grid* has two pure virtual methods. The first one is *init* that initializes the grid. The second method is *getInterval* that takes a point as input ($E$) and returns the bin number ($KE$) where the point $E$ is located, the transformed value of $E$ ($XEL$), the bin position of $E$ as a real number ($XE$) and the difference between $XE$ and $KE$ ($XEK$), i.e. the fractional part of $XE$ calculated as $XE - KE$, used for interpolation purposes.

It is very important to notice that the original PENELOPE uses a logarithmic grid for the interaction cross sections that cannot be changed for consistency reasons (for further details, see sections 1.2.4 and 7.1.1 of the PENELOPE manual). Indeed in PenRed, like in PENELOPE, energy-dependent quantities are tabulated for a logarithmic grid of 200 energies, whose span the complete energy range in the simulation. At intermediate energies, these quantities are obtained by linear log-log interpolation. However, other grids can be used in physics, materials or tallies classes implemented by the user.

### 4.1.5 Materials

**src/kernel/materials**

PenRead reads the required physical information about each material such as interaction cross sections, relaxation data and physical properties, from material data files. These files, for the implemented PENELOPE physics case, are exactly the same as the used by the original PENELOPE code system.

Therefore, a material class, which is basically a database, has been implemented in order to store all the necessary material information to perform the simulations. Additionally to physical information, any material store also a very important simulation parameters. These are the particle absorption or cutoff energies for each particle type. In fact, when the energy of a particle is lower than its cutoff energy in a given material, it is supposed to be absorbed. Absorption energies are stored, in units of eV in an array called $EABS$, which size is the number of implemented particle types.

The base material class, *abc_material*, is defined at src/kernel/includes/pen_classes.hh, which includes a minimal interface that all materials must provide. That is, an array with absorption energies in $eV$ ($EABS$), material density in $g/cm^3$ ($DEN$) and its inverse ($RDEN = 1/DEN$). In addition, abc_material provide a list of basic methods, which are listed below.

- **initDone**: Returns the value stored at variable *initialized.* This variable should be stored if the material has already been initialized.
- **getEABS**: Taking a number as argument, returns the energy absorption of the particle which *kpar* index corresponds to the introduced number.
- **setEABS**: Take a *kpar* index and an absorption energy as arguments to set the corresponding $EABS$ value.

- **setDens**: Sets the material density (*DEN*) and its inverse (*RDEN*).
- **readDens**: Returns the material density.
- **readIDens**: Returns the inverse of material density.

Although *abc_material* class doesn't depend on any other PenRed component, some others depend on it, such as contexts, interactions etc. So, to ensure the compatibility with all PenRed components, new materials should be implemented as derived classes of *abc_material*. An example of implemented material can be found at "pen_material" files.

In addition, "pen_materials.hh" and "pen_materials.cpp" files include all header and source files of material implementations. Each new material created by the user must be included at previous files to be used within PenRed environment.

#### 4.1.6 Contexts

#### src/kernel/contexts

Context classes are intended to wrap all the required information to perform the particles simulations. For that purpose, an abstract template class (*abc_context*) has been defined at file *src/kernel/includes/pen_classes.hh*.

This template class takes a single type argument, which specify the base material class compatible with the context. Thus, contexts will be compatible only with this material class. However, as a derived class contains the base class, a context will be compatible too with all derived materials from the base material type. To exemplify this property lets suppose that we have a base material class *Material_A* with a derived material class *Material_B*. Then, we define a context type (*Context_A*) compatible with *Material_A* type. Finally, we can use *Material_B* type to instantiate the context materials using the template context method *setMats*, which gets a material type as template argument. Notice that *setMats* will not be defined for types which are not a derived class of base material class. This example is summarised at code 2.

```
//Base material definition. Material_A is derived
//directly from abstract material class "abc_material"
class Material_A : public abc_material{ ... };
//Derived material definition
class Material_B : public Material_A{ ... };

//Context definition selecting "Material_A" as base material type
class Context_A : public abc_context<Material_A>{ ... };

int main(){
...

//Set number of materials
size_t nMats = 5;
//Create the context
Context_A context;
//Set material type "Material_B" for our context
context.setMats<Material_B>(nMats);

//Instead, to use base class material we should use
//context.setMats<Material_A>(nMats);
...
}
```

Code 2: Context material compatibility example

Like materials, PenRed contexts should be created as derived class of *abc_context* to ensure compatibility with other components. The basic context variables defined at *abc_context* are listed below,

- **maxEABS**: This pointer stores the absorption energies of each geometry body. Geometries, as we will show later, must implement one method to obtain the absorption energy of each body and another one to get the corresponding material index. As both, materials and bodies, have a associated absorption energy, *maxEABS* will store the most restrictive value i.e. the lowest energy. During the simulation, the particles will "ask" to the context its local absorption energy to stop the simulation if necessary.
- **geoBodies**: Stores the number of bodies in the assigned geometry.
- **geometry**: An immutable pointer to geometry.
- **matsSet**: Gets **true** if materials have been created using the *setMats* method, otherwise takes **false**.
- **materials**: Array of pointers to materials used at simulation. Pointer type is specified at the template argument and the dimension of the array is determined by the constant *MAXMAT*.
- **nMats**: Stores the number of created materials, whose pointers are saved at *materials* array.

Also, a set of functions to manage materials array and geometry variables are provided,

- **getMatBaseArray**: Fills an array of constants pointers with the type *abc_material* with the material pointers stored at *materials* array.
- **getMatEABS**: Returns the absorption energy of specified material (*imat*) and particle index (*kpar*).
- **getEABS**: Returns the body absorption energy with index *ibody* for particle index *kpar*.
- **getDET**: Returns the detector index of the body with index *ibody*.
- **getDSMAX**: Returns the maximum distance to travel in the body with index *ibody*.
- **setGeometry**: Takes a pointer to geometry as argument and set it as context geometry. Also, fills *geoBodies* and allocate memory for absorption energies.
- **readGeometry**: Returns a constant pointer to actual geometry.
- **updateEABS**: Fills *maxEABS* array using stored geometry and materials.
- **setMats**: This template function takes one type as template argument (*derivedMat*) and a number (*M*) as function argument. The function will instantiate as many materials as specified by *M*. On success, material pointers will be stored at *materials* array and *nMats* stores the value of *M*. The type of instantiated materials will be the one specified by *derivedMat*. Notice that this type must be a derived class of context base material. So, a context can use any material type derived from its base material. However, take into account that the maximum number of materials that can be instantiated is specified by the constant *MAXMAT*. This function can be called only one time to ensure simulation consistency, i.e. number of materials or its types can't change during the simulation. Notice that the context doesn't create any material until this function is called.

- **getNMats**: Returns the number of created materials.

- **setMatEABS**: Takes as arguments the material index *M*, particle index *kpar* and energy absorption *eabs*. With those values, sets the absorption material energy for the specified particle at material index *M*. Notice that material cuttoff energy could be overwritten by body absorption energy if it is more restrictive. Also, any change on material *EABS* value will not change or update the context stored values in *maxEABS* array. It is necessary to perform a call to *updateEABS* function to update maximum absorption energies.

- **readMaterial**: This template function takes a type as a template argument and a material index as a function argument. The function will return a constant reference to specified material casted to the specified type. If the material index is greater than the number of created materials (*nMat*) an exception will be thrown, otherwise the specified material pointer will be casted statically to the template argument type and deferred to return a constant reference.

  This function uses static_cast to avoid dynamic_cast overhead, however, this approach makes that function insecure. To ensure a correct usage, we use the following approach as function type returns,

  typename std::enable_if<std::is_base_of<baseMat, derivedMat>::value,
  derivedMat&>::type

  So, we can ensure that this function is used only to cast to derived types of base material, otherwise the code will not compile. Obviously it doesn't make the function safely, since could be specified a different or incompatible type than the one used at *setMats*. To check if a derived type is compatible with instantiated context materials, use the template function *compatible*.

- **getMaterial**: Same as *readMaterial* but returns a non constant reference to material i.e. a mutable object.

- **readBaseMaterial**: Takes a material index as argument and returns a constant reference with the context base material type to the specified material index. Unlike *readMaterial* or *getMaterial* this function doesn't perform any downside cast. If material index is greater than *nMats* an exception will be thrown.

- **getBaseMaterial**: Same as *readBaseMaterial* but returns a non constant reference to material.

- **compatible**: This template function, gets a type as template argument. Then, perform a *dynamic_cast* to try to convert the pointer of the material with index 0 to a pointer of the specified type. If the conversion has been done successfully, returns **true**, otherwise returns **false**. So, this function checks if instantiated context materials can be casted to the specified type. Notice that if *setMats* has not been executed successfully, all material pointers will be null pointers and any type checked will returns a non compatible type result.

A context type class example can be found in *pen_context* files. To implement new contexts, user must include the header and source files at *pen_contexts.hh* and *pen_contexts.cpp* respectively.

### 4.1.7 Particles

#### src/kernel/particles

Following with the same schema, PenRed provides a particles base class (*abc_particle*) as abstract template class defined in *src/kernel/includes/pen_classes.hh* file. To create a particle the following types must be specified using the template arguments,

- **stateType**: The particle state to be used (see section 4.1.2).
- **contextType**: The context class compatible with this particle.
- **materialType**: Which material needs this particle to be simulated.

In addition, any particle derived from *abc_particle* needs to provide, when constructor is called, some values to the *abc_particle* constructor. These values are, ordered, a compatible context, a particle index *KPAR* which should be included at the particle enumeration index (see section 4.1.7.1), the number of interactions and, if an annihilation can be done, the energy produced when it happens (*annihilationEDep*) i.e. the sum of the energy at rest, in $eV$, of both particle and antiparticle. For example, for positrons *annihilationEDep* is two times the electron rest energy (in $eV$). Code 3 shows the minimum required particle constructor definition, nevertheless, it can require more arguments depending on particle needs, as happens at PenRed built-in particles (examples can be found at *src/kernel/particles*).

```

//Define a particle class "particle_A" which uses as particle
//state type the "state_type" class, as a compatible context the
//"context_type" class and, finally, the "material_type" class as
//compatible material.
class particle_A : public abc_particle<state_type,context_type,material_type>{

...

public:

//Define the minimum class constructor
particle_A(const context_type& contextIn) : abc_particle(contextIn,
    PARTICLE_INDEX,NINTERACTIONS,0.0){...}

...

};
```

Code 3: Minimum particle constructor definition. The arguments required by the *abc_particle* constructor are, in order, a compatible context (*contextIn*), a particle numerical index (*PARTICLE_INDEX*), which should be constant and shared for all the instances of the same particle type, the number of interactions (*NINTERACTIONS*) and the annihilation energy produced at rest, which is zero for our particle because this kind of interaction is not allowed.

As we saw at section 4.1.6, a context has a base material type specified by its template argument. However, it can instantiate and store derived materials. To ensure than the particle is compatible with the introduced context, *abc_particle* constructor will call the *compatible* context function to check if *materialType* is compatible with the materials created by the context. If they are not compatible an exception will be thrown. A context cannot change its material number or type once they have been created, so, is safe to use the context function *readMaterial*.

Particle base class (*abc_particle*) contains the following common variables,

- **context**: A constant reference to the context used to create the particle.
- **kpar**: A numerical particle type identifier. This identifier is constant and is initialized at constructor call.
- **interactions**: Constant value that stores the number of available interactions for this particle.
- **annihilationEDep**: Energy produced on particle annihilation at rest (eV).
- **ELAST1**: Should store particle energy at last JUMP call.
- **P**: Is an array that stores the inverse mean free path of each allowed interaction. Must be computed at JUMP call to calculate the distance until next interaction.
- **ST**: Total inverse mean free path calculated as sum of all $P$ components.
- **XEL, XE, XEK, KE**: Grid variables for current energy (see section 4.1.4). Must be recalculated when energy changes.
- **P0**: Inverse mean free paths modified by interaction forcing methods.
- **LFORC**: Boolean array that stores if each interaction has been forced or not.
- **KSOFTI**: This index indicates if the particle is losing energy because soft interactions i.e. during the step. The value should be set at JUMP call. It takes a value of 1 if soft energy loss is required and 0 otherwise.
- **state**: Stores the actual state of the particle. The state type will be specified via the template argument *stateType*.
- **pmat**: Is a constant pointer that points to the current context stored material where the particle is moving. This pointer is automatically set if the main program uses the provided functions to move the particle, as we will see below. However, if the user adds his/her own functions, this pointer must be handled manually.
- **dsef, dstot, ncross**: These tree variables store the distance traveled in the origin material (non void zones), the total traveled distance including void zones and the number of interfaces crossed respectively at last *step* call. Like *pmat*, these variables will be automatically handled if the main program uses the provided move functions.
- **MATL, IBODYL, XL, YL, ZL**: Stores the last material and body indexes, and the position X,Y,Z before the last *step* call. Again, these variables will be handled automatically by provided move functions.
- **DEA**: Energy loss at last event.

Also, particle base class has a set of pure virtual methods that defines the behaviour of the particle and, for instance, must be implemented for each new particle. These methods are listed below,

- **START**: Will be called when the particle crosses an interface or begins the simulation. An interface cross should be managed by geometry step function and implies a material or detector indexes change. This will be explained in detail at geometry section 4.2.

- **JUMP**: Computes the distance until next interaction. That distance should be stored in $DS$ variable. Some particles need to limit traveled distance to use soft energy loses. This maximum distance is provided by $DSMAX$ variable.
- **JUMPF**: Same as $JUMP$ but using interaction forcing techniques.
- **KNOCK**: Selects, according to mean free paths, and computes the next interaction. Returns particle lost energy ($DE$) and the sampled interaction identifier ($ICOL$). Usually, secondary particles are produced by this function.
- **KNOCKF**: Same as $KNOCK$ but using interaction forcing techniques.
- **softEloss**: Computes the particle lost energy by soft interactions during the traveled distance. The position where the energy is deposited should be returned via the $X$, $Y$ and $Z$ variables. Parameter $DE$ should be filled with the total lost energy.
- **dpage**: This method actualizes the particles age. The time of flight is computed using the distance travelled in the current material ($dsef$) and the total travelled distance (*dstot*). A *dstot* greater than $dsef$ implies that the particle travelled a non zero distance in *void* material (material with index 0).
- **page0**: Initialize the required variables to compute particle time. Should be called before the simulation of a sampled particle begins.
- **annihilate**: Performs particle annihilation.

Finally, particle class includes some common auxiliary methods that are already implemented for all new particles,

- **DSef**: Returns *dsef* value.
- **DStot**: Returns *dstot* value.
- **NCross**: Returns *ncross* value.
- **updateMat**: Updates material pointer according to material index stored at particle state.
- **setMat**: Gets a material index as argument to change the stored index in particle state. Also, updates the material pointer *pmat* with a call to *updateMat*.
- **setStep**: Gets three parameters as arguments to update particles *dsef*, *dstot* and *ncross* values respectively.
- **lastMat**: Returns the material index stored at *MATL*.
- **lastBody**: Returns the body index stored at *IBODYL*.
- **lastPos**: Gets three doubles by reference to store the last position coordinates i.e. *XL*, *YL* and *ZL* respectively.
- **setLastMat**: Gets a material index as argument to update *MATL* value.
- **setLastBody**: Gets a body index as argument to update *IBODYL* value.
- **setLastPos**: Gets a three doubles as argument to update last position coordinates i.e. *XL*, *YL* and *ZL* respectively.

- **jumpVolume**: This method forces the particle to jump the current volume until next interface or geometry limit. Requires a geometry as argument to perform an “infinite step” ($10^{35} cm$), which should jump the current particle volume. Also, it stores the position, material and body before the step call in *XL*, *YL*, *ZL*, *MATL* and *BODYL* respectively. Finally, if required, update particles *PAGE* calling to *dpage* and material pointer *pmat* with a call to *updateMat*. This method can be called safely when particle is located in void materials.

- **move**: *move* method should be used to move the particle inside the geometry. As inputs, it takes a geometry instance (*geometry*), the distance to travel *ds*, and a random number generator *penRand*. On the other hand, as outputs, this method returns the deposited energy because soft interactions *de* and the position where the energy has been deposited *softX*, *softY* and *softZ*. Just like *jumpVolume*, *move* method handles and updates correctly material pointer *pmat* and the variables that store the information of last position. Also, if required, performs a call to *softEloss* to calculate the particle energy loss during the traveled distance. If there isn’t any soft energy loss, *de* will be set to 0 and *softX*, *softY* and *softZ* remain unchanged. Notice that this method expects an initial body with non void material i.e. material index must be greater than zero. To jump void volumes, use *jumpVolume* method. If this method is used to move the particle, changes in *MATL*, *BODYL*, *XL*, *YL*, *ZL*, *pmat* and *PAGE* variables are handled automatically.

- **xel**: Returns *XEL* value.

- **xe**: Returns *XE* value.

- **xek**: Returns *XEK* value.

- **ke**: Returns *ke* value.

- **getGrid**: Gets four variables by reference and fills they with *KE*, *XEL*, *XE* and *XEK* respectively which meanings have been discussed at section 4.1.4).

- **getEABS**: Returns the absorption energy of the body with index *IBODY* in the context geometry.

- **getDET**: Returns the detector index of the body with index *IBODY* in the context geometry.

- **getDSMAX**: Returns the maximum distance to travel inside the body with index *IBODY* in the context geometry.

- **reqSoftEloss**: Returns *KSOFTI* value.

- **getKpar**: Returns *kpar* value.

- **setBaseState**: Gets a particle base state variable (type *pen_particleState*) and copy it to the particle *state* variable.

- **setState**: Gets a particle state of type specified at *stateType* as input and copy it to particles *state* variable.

- **getBaseState**: Returns a mutable reference to particle *state* casted to *pen_particleState* type.

- **getState**: Returns a mutable reference to particle *state*.

- **readBaseState**: Returns a constant reference to particle *state* casted to *pen_particleState* type.

- **readState**: Returns a constant reference to particle *state*.

At *kernel/particles* folder, the user can found the three PENELOPE particle implementations: electron, gamma and positron.

#### 4.1.7.1 Adding particles

To add particles in PenRed, the user must includes the header and source files of his particle implementation in *src/kernel/particles/includes/pen_particles.hh* and *src/kernel/-particles/source/pen_particles.cpp* respectively. Also, adding the particle *kpar* identifier to *src/kernel/particles/includes/pen_particles_ID.hh* file before the last element of enumeration and assign a particle name in the method *particleName* in the same file. This will increment automatically the number of particle types stored in *nParTypes* constant. The enumeration identifier is supposed to be used as argument for the *abc_particle* constructor. The procedure to add a particle identifier and name to PenRed is shown at code 4.

```

//Particle index enumeration
enum pen_KPAR{
  PEN_ELECTRON,
  PEN_PHOTON,
  PEN_POSITRON,

  NEW_PARTICLE, // <---- Add a identifier of our new particle

  ALWAYS_AT_END // <---- Never add an identifier after that
};

//Particles index to string function
inline const char* particleName(const unsigned kpar){
  switch(kpar){
  case PEN_ELECTRON: return "electron";
  case PEN_PHOTON: return "gamma";
  case PEN_POSITRON: return "positron";

  //Add a text name for our particle
  case NEW_PARTICLE: return "newName";

  default: return nullptr;
  }
}
```

Code 4: Example of adding a particle to the existing particle enumeration

### 4.1.8 Interactions

To take some order on interactions, a base class for particle interactions has been created as abstract template class (*abc_interaction*) (defined at *src/kernel/includes/pen_classes.hh*). This one takes three types as template arguments, which corresponds to the particle, context and material types respectively. The only variable at the base class is *ID* which should store a unique interaction identifier for the corresponding particle. We encourage to use an enumeration to assign interaction indexes, as we done at section 5.4.

All interactions implemented as derived classes of the provided interface, must implement the following methods,

- **init**: Initializes interaction parameters taking a context as input.
- **iMeanFreePath**: Computes the particle inverse mean free path for this interaction at current material and energy.
- **interact**: Performs the interaction and returns the particle energy lost and the interaction $ID$.
- **interactF**: Same as *interact* but using interaction forcing techniques.

Notice that, currently, no other classes take or requires interaction base class or its derived as function or template arguments. So, this class is only a recommended pattern for the shake of readability, but not use it will not cause any compatibility issue. For example, the interactions can be implemented directly in the $KNOCK$ function.

Several examples of implemented interactions using this interface can be found at the corresponding source files of PenRed built-in particles.

## 4.2 Geometries

**src/geometry**

All PenRed geometries must be organised with bodies that are formed by a single material. However, the definition of these bodies depends on the geometry type. For example, PenRed includes two major geometry types. The first one, supposes that bodies are a single element, like quadric geometries. The second, uses a mesh to define the geometry, and each body is defined by many of these mesh elements. Notice that a material can be used by several bodies. Also, the user must take into account that material index 0 is reserved for *void* material i.e. the particles fly without interact at this regions. However, the body index 0 has not a special meaning, thus corresponds to the first defined body in the geometry.

In addition, bodies can be grouped to create detectors, which function will be explained below.

PenRed provide a abstract or interface class to implement custom geometries (*wrapper_geometry*), which includes the following common variables,

- **configStatus**: Stores the state of configuration step.
- **name**: Geometry identifier name.

and requires an implementation of the following methods:

- **getEabs**: Taking a body and a particle index as input parameters, returns the corresponding energy absorption.
- **getDSMAX**: Gets a body index as input parameter and returns the maximum distance allowed to travel in this body. This value should be used as input parameter for JUMP particles functions (*DSMAX*).
- **getDET**: Gets a body index as input parameter and returns the detector identifier of this body. If the body is not part of any detector, the expected returned value is 0.
- **getMat**: Gets a body index as input parameter and returns the material that fills this body. If the body is filled with void, the expected returned value is 0.

- **getElements**: Returns the number of elements in the geometry. This value could not match with the number of bodies. For example, for built-in voxel geometries, the number of elements corresponds to the number of voxels and the number of bodies is the same that the number of used materials.

- **getBodies**: Returns the number of bodies in the geometry. If the geometry doesn't use identifiable objects, we suggest to identify each material as a body. This trick has been used on built-in voxel geometries. This approach allows to use all tallies with any geometry.

- **getIbody**: Taking a text as input, this method returns a body identifier. Geometries must provide a method to identify its objects with text. For example, in quadric geometries, each body has a unique alias. At voxel geometries, the introduced text is converted to an integer value and is interpreted as material index. So, "1" corresponds to body index 0, "2" to body 1 and so on, because voxels with void material are not allowed.

- **locate**: Takes a base particle state section 4.1.2 as argument and updates the material and element index (IBODY) where particle is located. Also, if particle is outside of geometry system, locate must set IBODY index to a value greater or equal to the number of geometry bodies, i.e. the value returned by *getBodies* function.

- **step**: Moves the particle the specified distance (*DS*) or until an interface has been reached. Any material or detector index change must be interpreted as interface. Also, if behind the interface the particle find a void region (materials with index 0), the step function must jump this one until the next interface with a non void material region. If any non void region is reached at the particle direction, the geometry will consider that this particle escapes from the system. At this case, the particle will be moved an *infinite* distance (typically $10^{30} cm$).

  In addition to particle movement, the step method will record the total traveled distance in original material (*dsef*) and the total traveled distance (*dstot*), i.e. the sum of *dsef* and the distance traveled at void regions.

  Finally, if an interface has been crossed during the step, *ncross* must take a value different than zero. Notice that the particle state must be updated with the new position (*X*, *Y*, *Z*), and new body and material indexes (*IBODY* and *MAT*). Also, if the step receives a state with an *IBODY* index greater or equal to geometry bodies (value returned by *getBodies*) this must be interpreted as the particle comes from outside the geometry. Thus, *dsef* and *dstot* will store the same value, because the original material was void.

  We would like to emphasize the fact that the particle must be stopped if some interface has been crossed, which means that the particle changes of material or detector index. So, a single step call should move the particle travelling within only one material, the one where step begins, and possible void volumes. An example of a very basic *step* structure can be found at the code 5.

```
  void myGeometry::step(pen_particleState& state,
                        double DS, double &DSEF,
                        double &DSTOT, int &NCROSS) const{

      //Check if the particle comes from outside of geometry
      if(state.IBODY >= getBodies()){
          //Particle come from outside the system,
          //DS value is ignored
```

```
            moveToGeometry(state,DSEF,NCROSS);
            DSTOT = DSEF;
            return;
        }

        //The particle is in the geometry system

        //Get the distance until next interface and the
        //corresponding body and material indexes
        unsigned newIBODY, newMAT;
        double dsToInterface = nextInterface(state,
                                             newIBODY,
                                             newMAT);

        if(dsToInterface > DS){
            //The particle will be moved freely
            state.X += state.U*DS;
            state.Y += state.V*DS;
            state.Z += state.W*DS;

            DSEF = DS;
            DSTOT = DSEF;
            NCROSS = 0; //No interface crossed
            return;
        }
        else{
            //The particle find an interface
            DSEF = dsToInterface;
            NCROSS = 1;   //An interface crossed
            //Check if the material behind is void
            if(newMAT == 0){
                //Is a void region, find next interface or
                //escape from geometry
                unsigned newIBODY2, newMAT2;
                double dsToNonVoid = nextInterface(state,
                                                   newIBODY2,
                                                   newMAT2);

                if(newIBODY2 < getBodies()){
                    //The particle has crossed another
                    //interface and, for instance,
                    //remains in the geometry system
                    NCROSS += 1;
                }

                DSTOT = DSEF + dsToNonVoid;

                state.X += state.U*DSTOT;
                state.Y += state.V*DSTOT;
                state.Z += state.W*DSTOT;
                state.IBODY = newIBODY2;
                state.MAT = newMAT2;
            }
            else{
                //Is a non void region
                DSTOT = DSEF;
                state.X += state.U*DSEF;
                state.Y += state.V*DSEF;
                state.Z += state.W*DSEF;
                state.IBODY = newIBODY;
                state.MAT = newMAT;
            }
            }

        }

```

Code 5: Basic *step* implementation structure

- **configure**: Takes a configuration section (see section 3) to initialize the geometry.
- **usedMat**: Returns what material identifiers are used at current geometry.

Notice that previous methods only need the *pen_particleState* as particle state type. So, all geometries following this pattern will be compatible with any particle regardless its state type.

This "basic" geometry has been provided via the class *wrapper_geometry* defined in *src/kernel/includes/pen_classes.hh* file. However, with the aim of simplify new geometries implementation, two derived classes has been created, *abc_geometry* and *abc_mesh*. These classes, provide a partially predefined interfaces for geometries based on "objects" and "meshes" respectively. Both of them, implement all required geometry methods except *locate*, *step* and *configure*, which must be implemented by the user to create a new geometry. Nevertheless, a new geometry can be implemented directly as derived class of *wrapper_geometry*. Following sections will describe the characteristics of these two geometry types.

### 4.2.1 Objects based geometries

#### src/geometry/objects

*abc_geometry* provides a framework to implement geometries that use bodies as base element.

#### 4.2.1.1 Bodies

In *abc_geometry* class, bodies are the elemental unit to construct the geometries. All body types used at derived classes of *abc_geometry* should be created as derived classes of the provided base body class (*pen_baseBody*). This one contains the following basic member variables,

- *Mater*: Stores body's material index.
- *KDET*: Stores body's detector index.
- *localEABS*: Array with cuttoff absorption energies for each particle type (with dimension *constants::nParTypes*) in the body. This value will be used by the context to determine the most restrictive absorption energy.

Although is not mandatory to create a specific geometry body class as a derived one of *pen_baseBody*, we strongly recommend extending this base class if the user needs to implement a new geometry with custom bodies.

By default, all energy local absorption will be set to zero. So, all particle entering in a uninitialized body will get the material energy absorption as the cuttoff energy, because will be more restrictive.

#### 4.2.1.2 Body based geometries

Body geometries are implemented in PenRed, like kernel components, via an abstract template base class named *abc_geometry*, which derives from *wrapper_geometry*. In this case, only one template argument must be specified, corresponding to the body type. Each geometry has, in addition to base class (*wrapper_geometry*) variables and methods, an array of bodies (*bodies*) with dimension defined by $pen_geoconst :: NB$, and the corresponding body counter ($NBODYS$). Also, by default, the class will check on construction call if the specified body type is a derived class of *pen_baseBody*. If it doesn't derives from *pen_baseBody*, the constructor will throw an exception. This check can be avoided setting the constant *checkBody* to false.

In respect to member methods, derived classes of *abc_geometry* must implement three main methods: *locate*, *step* and *configure*. These methods define the behaviour of the geometry, and have been explained in *wrapper_geometry* class at section 4.2. All other virtual methods are already implemented and ready to use. In addition, geometry base class has some auxiliary methods,

- **setBodyEabs**: Actualizes the list of absorption energies for specified body. If some component of $EABS$ input array is negative or zero, will be ignored and the energy absorption of this particle type will remain unchanged.

PenRed quadric geometry implementation can be found at *geometry* folder.

### 4.2.2 Mesh geometries

**src/geometry/meshes**

As body based geometries, mesh geometries uses a basic mesh element structure as mesh unit. The base type for this structures is defined at *pen_baseMesh*, but is simpler than the used on body based geometries. In fact, only defines one variable, *MATER*, which stores the material index of the mesh element.

New mesh based geometries should be implemented as derived classes of *abc_mesh*, which is a generic template class for geometries that uses meshes. Like body based geometries, this template class gets the type of the mesh element (*meshElement*) via the template argument and check if it is a derived type of *pen_baseMesh*. Also, derives from *wrapper_geometry*, so is compatible with all particle and other PenRed components.

This kind of geometries don't use objects, but, to emulate objects, these geometries consider each material as single object. To implement this approach, *abc_mesh* class has two arrays (*DSMAX* and *KDET*) to store the maximum allowed distance to travel and the detector number of each material (and body) respectively. Notice the lack of an array to store the absorption energies for each material/body. This is because it has no sense to set a different absorption energy for bodies and materials, since each material creates a single body. So, a call to *getEabs* with any input will return $1^{-15}eV$. On the other hand, the variable *nBodies* will store the number of bodies i.e. the number of used materials.

To handle the mesh, *abc_mesh* has a *mesh* pointer to construct an array of *meshElement* elements, a variable to store the mesh dimension (*meshDim*) and another one to store the number of mesh elements *nElements*. Notice that mesh dimension and number of elements could mismatch because represent the number of allocated and used elements respectively. To handle mesh creation and destruction two protected methods have been implemented,

- **clearMesh**: Frees the allocated memory to store the mesh and set *meshDim*, *nElements* and *nBodies* to zero. In addition, mesh pointer is set to null after deallocation and *meshStatus* to uninitialized mesh.

- **resizeMesh**: After a call to *clearMesh*, allocates memory to store as many elements as specified. The *meshDim* value will be set to the specified dimension, but not *nElements* and *nBodies*, whose values will be 0. This is because the mesh has not been initialized, only allocated. Thus, we don't know the number of mesh elements nor bodies used at the geometry.

Notice that the mesh will be automatically cleared at class destructor using the *clearMesh* method. So, is not necessary that the user handles memory allocations/deallocations.

In the same way as in bodies based geometries (sec 4.2.1.2), only *locate*, *step* and *configure* methods must be implemented.

#### 4.2.2.1 Geometry creation

To show how to create a new geometry, PenRed provides a completely dummy body based geometry which never moves the particle. However is very simple and illustrative. First, lets see its header file at code 6.

```
#ifndef __PEN_DUMMY_GEOMETRY__
#define __PEN_DUMMY_GEOMETRY__

#include "geometry_classes.hh"

class pen_dummyGeo : public abc_geometry<pen_baseBody>{
 DECLARE_GEOMETRY(pen_dummyGeo)

  public:
  pen_dummyGeo() {
    configStatus = 0;
  }

  int configure(const pen_parserSection& /*config*/,
                unsigned /*verbose*/){
    return 0;
  }
  void locate(pen_particleState&) const;
  void step(pen_particleState&,
      double,
      double &,
      double &,
      int &) const;

  inline unsigned getIBody(const char* /*name*/) const {
    return getElements();
  }
};

#endif
```

Code 6: Dummy geometry header file

Lets see the code in detail. First of all, we found the unique necessary include file *geometry_classes.hh*, which includes all the required definitions to construct geometries. Following line shows the class declaration *pen_dummyGeo* which derives from *abc_geometry* and uses the *pen_baseBody* type as body type. The next line is possibly the most "strange". We are referring to,

*DECLARE_GEOMETRY(pen_dummyGeo)*

This calls a macro named *DECLARE_GEOMETRY*, which definition can be found at *geometry/includes/pen_geometry_register.hh* file and is summarised at code 7.

```
#define DECLARE_GEOMETRY(Class) \
  public: \
  inline int registerStatus() const { return ___register_return;} \
  inline const char* readID() const { return ___ID;}\
  private: \
  static const char* ___ID;\
  static const int ___register_return;
```

Code 7: Declare geometry macro

Basically, this macro adds some auxiliary variables and methods to our class definition that will allow us to instantiate our new geometry using a text identifier. To achieve that purpose, we need another macro, which will be called at the source file. At this point, we only need to know that **the macro argument must be the class name**.

So, following with the header file, the class constructor sets the *configStatus* to 0. This value indicates that the geometry is uninitialized. Then, *configure* method gets a *pen_parserSection* (see section 3), with all the required data for the initialization, and a verbose level. Our dummy function doesn't requires any configuration, so, both arguments are ignored. At a "normal" geometry definition, the verbose index should be interpreted as following,

- 0: Any message should be printed.
- 1: Only error messages should be printed.
- 2: Errors, warnings and important information should be printed.
- $> 2$: Prints all available information.

In addition, notice that geometry configuration must return a 0 if the configuration has been done successfully. Otherwise, a non zero value should be returned. The following *locate* and *step* methods will be explained at source section. Finally, *getIBody* returns always the number of elements for the dummy geometry. This is because the dummy geometry has not any body, so is not possible to select a body using a text identifier (*name*).

Once we have seen the header file, lets turn to source file. This one is summarised at code 8.

```
#include "dummy_geo.hh"

void pen_dummyGeo::locate(pen_particleState& /*state*/) const{
}

void pen_dummyGeo::step(pen_particleState& /*state*/, double /*DS*/, double &
    DSEF, double &DSTOT, int &NCROSS) const{

  DSEF = 0.0;
  DSTOT = 0.0;
  NCROSS = 0;
}

REGISTER_GEOMETRY(pen_dummyGeo,DUMMY)
```

Code 8: Dummy geometry source file

First, the header file discussed above is included. Secondly, we implement the *locate* method. In our dummy case, the particle can be in any body, so *locate* does nothing. A normal *locate* method should change the state variables *IBODY* and *MAT* according to the state position *X*, *Y* and *Z*.

Following, we found the *step* implementation. This one doesn't move the particle, so sets the output variables *DSEF*, *DSTOT* and *NCROSS* to zero with no changes on the particle state (*state*). A more generic and useful implementation pattern of the *step* method has been discussed at code 5.

Finally, we found the last strange thing, another macro,

*REGISTER_GEOMETRY(pen_dummyGeo,DUMMY)*

The implementation of this macro is summarised at code 9.

```
#define REGISTER_GEOMETRY(Class, ID) \
  const int Class::___register_return = penGeoRegister_add<Class>(static_cast<
    const char *>(#ID)); \
  const char* Class::___ID = static_cast<const char *>(#ID);
```

Code 9: Register geometry macro

Basically, this macro registers the geometry *Class* type with the text identifier *ID*. Then, our geometry can be instantiated using this text identifier and the function shown at code 13. This mechanism is described in detail at the string instantation mechanism (section 4.3).

Finally, to add our geometry in the compilation process, we must to include our header file *dummy_geo.hh* to *geometry/objects/includes/pen_object_geos.hh* and the source file *dummy_geo.cpp* to *geometry/objects/source/pen_object_geos.cpp*, as shown at codes 10 and 11 respectively.

```
#ifndef __PENELOPE_OBJECT_GEOMETRIES__
#define __PENELOPE_OBJECT_GEOMETRIES__

//Include implemented geometries include files
#include "quadric_geo.hh"
#include "dummy_geo.hh"    // <---- Our new geometry

#endif
```

Code 10: Geometry include file

```
//Include implemented geometries source files
#include "quadric_geo.cpp"
#include "dummy_geo.cpp" // <---- Our new geometry
```

Code 11: Geometry source file

After this few steps, our geometry will be compiled at the next PenRed compilation. So, at this section, we have shown how to create and include a user defined geometry to PenRed package. The same procedure can be applied to mesh based geometries changing the base classes of which our geometry derives. Also, for meshes, the include and source files must be added to *geometry/meshes/includes/pen_mesh_geos.hh* and *geometry/meshes/-source/pen_mesh_geos.cpp* respectively. Finally, if we doesn't use any of the two provided geometry patterns, we should derive our geometry class directly from *wrapper_geometry* and add our include and source files to *geometry/includes/pen_geometries.hh* and *geometry/source/pen_geometries.cpp* respectively.

## 4.3 String instantiator mechanism

### src/lib/instantiator

To provide flexibility to PenRed engine, we implement a mechanism to instantiate derived classes using a text identifier. With this method, PenRed can instantiate any types of sources, geometries, tallies, etc. on runtime at the main program without changing the code source.

This approach is achieved via a registration class. This one is a template class which gets a base type as template argument (*motherClass*). The key of this register is that only types which are convertible to the base type could be registered. The definition of the register class can be found at *src/lib/instantiator/instantiator.hh*.

That register class is named *instantiator*, and usually is instantiated using a static variable to be accessible along the code. For instance, we have ensured that all its methods are thread safe.

Following we will discuss how PenRed allows to instantiate a derived type using a *instantiator* class. First, we need to register available types, which must be convertibles to *motherClass*. To do that, we need a function that gets two types and tries to cast dynamically a pointer of second type to the first one. This function is showed at code 12.

```
template <class mother, class sub>
void instanceInheritance(mother*& pmoth){
  sub* psub = new sub;
  pmoth = nullptr;
  pmoth = dynamic_cast<mother*>(psub);
  if(pmoth == nullptr){
    delete psub;
  }
}
```

Code 12: Instantiator template function

That function instantiates the specified class type *sub* and, then, checks if this type can be convertible to *mother* type. So, we already have a mechanism to instantiate a “derived” type and return a “base” or “mother” type ensuring that both types are compatibles. The next step is to save this function with the properly template arguments in the register. Fortunately, we can save it using a *std::function* with the signature,

$$std :: function < void(motherClass * \&) > \tag{1}$$

This standard C++ wrapper can store, copy, and invoke any callable target. In our case, the target is the function,

$$\begin{gathered} template < classmotherClass, classsubclass > \\ void\ instanceInheritance(motherClass * \&pmoth) \end{gathered} \tag{2}$$

This callable object is stored in a map where each value pair has the types,

$$< std :: string|std :: function < void(motherClass * \&) >> \tag{3}$$

where *std::string* is a standard C++ string. Now, with the properly string, we can instantiate the corresponding registered type, as shown at code 13.

```
  motherClass* createInstance(std::string typeID) {

  //Lock the mutex
  std::lock_guard<std::mutex> guard(lock);

  //Search specified ID
  if(creatorsMap.find(typeID) != creatorsMap.end()){
    motherClass* pmoth = 0;
    creatorsMap[typeID](pmoth);
    return pmoth;
  }

  return nullptr;
}
```

Code 13: Instantiate function via text identifier

With this approach, we don't need to add any instantiating method on base classes nor derived classes. Also, the approach, allows to register not only derived classes, but also types convertible to *motherClass*.

The implemented methods at *instantiator* class are,

- **registred**: Returns the number of registered classes.
- **addSubType**: As function argument, gets a constant char pointer (*typeID*), which will be the identifier name. Also, the type to register must be specified as a template argument (*subclass*). This method tries to register the specified type *subclass* with the name *typeID* as key in the map. If specified type is not convertible to the *instantiator* template argument type (*motherClass*), a non zero value will be returned. On success, the function *instanceInheritance<motherClass, subclass>* will be stored in the map with the specified identifier name. Notice that each name must be unique in each register.
- **createInstance**: Creates an instance of the registered type according to the introduced name. Then, returns a pointer to the instantiated element casted to the *instantiator* template argument type (*motherClass*). If the name does not exists in the register, a null pointer will be returned.
- **typesList**: Returns a string with all registered names.

This technique has been used in PenRed code to facilitate the addition of tallies, geometries and particle sources. Notice that this method is not optimal regarding in execution time. However, is only used during initialization, thus its performance impact is negligible. Also, the registration mechanism for new modules is handled automatically by the provided macros, as we saw at geometry creation section (section 4.2.2.1). With this approach, we try to simplify its usage to the user. Furthermore, the register is performed before the main beginning using static variables, which could produce much errors if is not handled correctly. For example the well know *static initialization order "fiasco"*. The provided macros handle correctly these effects and the user shouldn't take care about that.

## 4.4 State samplers

### src/particleGen

This section will explain how *State Samplers* work. The objective of this component is to sample the initial particle states. For that purpose, PenRed uses the following different sampler types,

- **Time**: Samples initial particle age (variable *PAGE*). This is an **optional** sampler type.

- **Spatial**: Samples initial particle position $X$,$Y$,$Z$. $IBODY$ and $MAT$ identifiers will be determined by sampled position.
- **Direction**: Samples initial particle direction vector ($U$,$V$,$W$).
- **Energy**: Samples initial particle energy $E$.
- **Specific**: This sampler type is implemented as a template class with a single argument. This one specify the particle state type to be sampled. Unlike previous samplers, specific samplers gets the whole particle state and can modify all its variables.

Take into account that the samplers will be called following the order listed at the above list. Therefore, time, position, energy or direction samplers could be overwrote by specific samplers. At the following sections all sampler types and how they are interrelated will be explained.

### 4.4.1 Spatial Samplers

**src/particleGen/spatial**

As its name suggest, spatial samplers handles the particle position sampling step i.e. sets the $X$, $Y$ and $Z$ variables of the particle states. These samplers must be created as derived class of *abc_spatialSampler*, which provide a common interface for all spatial samplers. The base class contains a rotation matrix (*rotation*) and a translation vector (*translation*) as member variables. These variables store the rotation and the translation applied to the sampled position respectively. So, the final position will be obtained by,

$$X' = T + R \cdot X, \tag{4}$$

where $X$ is the initial sampled position, $T$ the translation vector, $R$ the rotation matrix and $X'$ the final position. All new spatial samplers must implement the method to create the initial $X$ defining the pure virtual method *geoSampling*. Also, is required to implement the pure virtual method *configure*, which takes as argument a *pen_parserSection* variable (section 3) to configure the sampler. In addition, to facilitate the creation of the rotation matrix, *abc_spatialSampler* implements the function *setRotationZYZ*, which takes as arguments three angles and create the rotation matrix as the product of three rotations around the axis $Z$, $Y$, $Z$,

$$R(\alpha_1,\alpha_2,\alpha_3) = R_z(\alpha_1)R_y(\alpha_2)R_z(\alpha_3) \tag{5}$$

To exemplify how to implement spatial samplers, the codes 14 and 15 show how to implement a basic box source. First, on the header file, we found a typical derived class definition where *box_spatialSampling* inherits from *abc_spatialSampler*. Immediately inside the class definition, we use the macro,

*DECLARE_SAMPLER(box_spatialSampling)*

which is very similar to the macro used to include new geometries at PenRed engine (section 4.2.2.1). As for geometries, this macro wraps the declaration of some necessary variables and functions to simplify the incorporation of the sampler in the PenRed system. Following, we found the specific variables required for this sampler. These are, *dx*, *dy* and *dz*, which store the box sizes on $X$, $Y$ and $Z$ axis respectively.

Finished the variables declaration, follows the class constructor. In this case, the constructor only initializes the sampler with some default values. Then, we find the definitions of the two mandatory functions that must be implemented i.e. *geoSampling* and *configure*. Notice at this point that *geoSampling* is a **constant** member function, which means that the state of the any class derived from *abc_spatialSampler* can't change during the sampling call. This restriction is used to avoid race-conditions or other inconsistencies on multi-threading simulations. So, the state of our spatial sampler class must be set at the *configure* call.

```

#ifndef __BOX_SPATIAL_SAMPLING__
#define __BOX_SPATIAL_SAMPLING__
class box_spatialSampling : public abc_spatialSampler {
  DECLARE_SAMPLER(box_spatialSampling)

private:

  double dx, dy, dz;

public:

  box_spatialSampling() : dx(1.0),
                          dy(1.0),
                          dz(1.0)
  {}

  void geoSampling(double pos[3], pen_rand& random) const;

  int configure(const pen_parserSection& config,
                const unsigned verbose = 0);

};

#endif
```

Code 14: Spatial sampler header file example

Once discussed the header file, lets turn on our sampler source file. After including the header file, we find the sampling function *geoSampling*. This method takes the position to be sampled as a vector with three components ($X$,$Y$,$Z$) and a random number generator. Then, in this case, the position is sampled in a box of dimensions $dx \times dy \times dz$ cm using the equation 6 for each coordinate,

$$x = dx \cdot \xi - dx/2 = dx\,(\xi - 0.5) \tag{6}$$

where $\xi$ is a random value in the interval $[0, 1]$. Next, function *configure* gets all the required values to configure the source i.e. $dx$, $dy$, $dz$ and the values $x, y, z$ for the box translation. Notice that the translation is stored at the variable *translation* defined at the base class *abc_spatialSampler*. Also *configure* checks that *config* variable contains all the required configuration data and if some one has a non valid value. On success, configuration function returns a 0, otherwise a non zero value will be returned. To finish with this source file, lets see the register macro,

*REGISTER_SAMPLER(box_spatialSampling,BOX)*

This macro takes as arguments the name of our spatial sampler class (*box_spatialSampling*) and a text identifier for that class (*BOX*). Then, as happens for geometries, this class is registered using the method described at section 4.3 and can be instantiated via its name identifier.

```
#include "box_spatialSampling.hh"

void box_spatialSampling::geoSampling(double pos[3], pen_rand& random) const{

  pos[0] = dx*(random.rand()-0.5);
  pos[1] = dy*(random.rand()-0.5);
  pos[2] = dz*(random.rand()-0.5);

}

int box_spatialSampling::configure(const pen_parserSection& config, const unsigned verbose){

  int err;

  err = config.read("size/dx",dx);
  if(err != INTDATA_SUCCESS){
    if(verbose > 0){
      printf("boxSpatial:configure:unable to read 'size/dx' in configuration. Double expected\n");
    }
    return -1;
  }

  err = config.read("size/dy",dy);
  if(err != INTDATA_SUCCESS){
    if(verbose > 0){
      printf("boxSpatial:configure:unable to read 'size/dy' in configuration. Double expected\n");
    }
    return -1;
  }

  err = config.read("size/dz",dz);
  if(err != INTDATA_SUCCESS){
    if(verbose > 0){
      printf("boxSpatial:configure:unable to read 'size/dz' in configuration. Double expected\n");
    }
    return -1;
  }

  if(dx < 0.0 || dy < 0.0 || dz < 0.0){
    return -2;
  }

  err = config.read("position/x",translation[0]);
  if(err != INTDATA_SUCCESS){
    if(verbose > 0){
      printf("boxSpatial:configure:unable to read 'position/x' in configuration. Double expected\n");
    }
    return -2;
  }

  err = config.read("position/y",translation[1]);
  if(err != INTDATA_SUCCESS){
    if(verbose > 0){
      printf("boxSpatial:configure:unable to read 'position/y' in configuration. Double expected\n");
    }
    return -2;
  }

  err = config.read("position/z",translation[2]);
  if(err != INTDATA_SUCCESS){
    if(verbose > 0){
      printf("boxSpatial:configure:unable to read 'position/z' in configuration. Double expected\n");
    }
    return -2;
  }

  if(verbose > 1){
    printf("Box center (x,y,z):\n %12.4E %12.4E %12.4E\n",translation[0],translation[1],translation[2]);
    printf("Box size (dx,dy,dz):\n %12.4E %12.4E %12.4E\n",dx,dy,dz);
  }

  return 0;
}

REGISTER_SAMPLER(box_spatialSampling,BOX)
```

Code 15: Spatial sampler source file example

Finally, to include our new sampler to the PenRed compilation, we must include our header and source file to *src/particleGen/spatial/includes/spatialSamplers.hh* (code 16) and *src/particleGen/spatial/source/spatialSamplers.cpp* (code 17) , as we done for new geometries with their corresponding files.

```
#ifndef __PENRED_SPATIAL_SAMPLERS__
#define __PENRED_SPATIAL_SAMPLERS__

#include "box_spatialSampling.hh"   // <---- Our new sampler
#include "point_spatialSampling.hh"

#endif
```

Code 16: Spatial samplers include files

```
#include "box_spatialSampling.cpp" // <---- Our new sampler
#include "point_spatialSampling.cpp"
```

Code 17: Spatial samplers source files

This procedure to create and include new spatial samplers to PenRed engine is analogous to the required to incorporate new samplers of any type. Only the sampling function and the files where include our headers and source files changes between sampler types. Also, for specific samplers, we must use a different macro, as we will see.

### 4.4.2 Direction Samplers

#### src/particleGen/direction

As spatial samplers, direction samplers have their own interface class to provide a common interface to this kind of samplers. This interface is defined at *abc_directionSampler* abstract class. Similarly, new direction samplers must define the *configure* method and a function to sample the direction vector $(U, V, W)$. This function is named *directionSampling*. The procedure to create new direction samplers is equivalent to the one shown for spatial samplers. Furthermore, to include a new direction sampler the same macros must be used.

The only difference with respect to spatial samplers is that the *directionSampling* function gets an array with three dimensions which must be filled with the sampled direction

$(U, V, W)$ instead of the position. Also, header and source files of new samplers must be included at *src/particleGen/direction/includes/directionSamplers.hh* and *src/particleGen/direction/source/directionSamplers.cpp* respectively.

### 4.4.3 Energy Samplers

#### src/particleGen/energy

Energy samplers use *abc_energySampler* as abstract base class. New samplers derived from the base class must define the methods *configure* and *energySampling*, which fill the *energy* variable with the sampled energy. However, these kind of samplers have a special property. The *abc_energySampler* class, has two public variables (*maxEnergy* and *minimumEnergy*) that define the allowed energy range for the implemented physical models. If the sampler returns an energy value out of this range, the program will throw an exception. In addition, the configuration method must return the maximum possible sampled energy filling the variable *Emax*. This variable will be used by material and context components to create the corresponding energy grids for the simulation.

The method to register new energy samplers is equivalent to the spatial samplers case. The header and source files must be included at *src/particleGen/energy/includes/energySamplers.hh* and *src/particleGen/energy/source/energySamplers.cpp* respectively.

### 4.4.4 Time Samplers

#### src/particleGen/time

Finally we have the time samplers. Analogous to other samplers, time samplers use *abc_timeSampler* as a base class. Also, new ones must implement the *configure* and *timeSampling* methods. In this case, *timeSampling* gets the variable *time* as argument, which must be filled with the sampled particle time.

Once again, we should register time samplers as we did for spatial samplers. Headers and source files must be added to *src/particleGen/time/includes/timeSamplers.hh* and *src/particleGen/time/source/timeSamplers.cpp* files respectively.

### 4.4.5 Specific Samplers

#### src/particleGen/specific

These samplers differ from the previous ones because are intended to sample all properties of a specific particle states, such as stoke parameters on polarized photons. They share a common interface defined at *abc_specificSampler*, which is an abstract template class that takes as template argument the particle state type compatible with the sampler. So, it can be used to implement a sampler for any user defined state. As previous types, new specific samplers must implement a configuration and sampler methods, however, these methods have differences compared with previous samplers.

First, configuration method (*configure*) takes some extra arguments compared with other sampler types, as shown at code 18. The first one is *Emax*, which is equivalent to the same variable used at energy samplers configuration. Then, *configure* gets a constant pointer of each generic sampler type i.e. spatial, direction, energy and time. Thus special samplers can use generic samplers to delegate, partially, the state sampling.

```
int configure(double& Emax,
      const abc_spatialSampler* pSpatial,
      const abc_directionSampler* pDirection,
      const abc_energySampler* pEnergy,
        const abc_timeSampler* pTime,
        const pen_parserSection& config,
        const unsigned verbose);
```

Code 18: Configure method for specific samplers

Respect to the sampling method (*sample*) shown at the code 19, the first notable difference is the fact that this method is not constant. This means that *sample* method could change the state of the sampler class. This approach allows us to create versatile and completely generic samplers but, in return, we need to ensure that the *sample* method is thread-save since is not ensured by the constant restriction. Next differences appear at the function argument variables. Leaving aside the random generator, it gets a reference to the whole particle state, so this method can sample all state properties. In addition, the argument *genKpar* must be set with the particle index (see section 5.4) that will be generated by the sampling method. Notice that, as consequence, specific samplers could sample different particles. Finally, the last variable to take into account is *dhist*, which must be filled with the history increment. For most samplers, this increment is always 1, however, some samplers should require to skip more than a single history between sample calls or even zero histories. For example, if the sampler uses a splitting variance reduction technique, successive calls to *sample* could produce particles belonging to the same history.

```
  void sample(pen_particleState& state,
          pen_KPAR& genKpar,
          unsigned long& dhist,
          pen_rand& random);
```

Code 19: Sample method for specific samplers

As we mentioned before, spatial samplers can delegate partially or totally the sampling of the generic part of the particle state. Thus, we need a mechanism to specify what generic samplers requires our specific sampler. This is done using the *abc_specificSampler* constructor, which gets a *__usedSamp* enumeration value as argument. This enumeration is summarised at the code 20.

```
enum __usedSamp{
        USE_SPATIAL    = 1 << 0,
        USE_DIRECTION  = 1 << 1,
        USE_ENERGY     = 1 << 2,
        USE_TIME       = 1 << 3,
        USE_GENERIC    = 1 << 4,
        USE_NONE       = 1 << 5
};
```

Code 20: Specific sampler usage flags

We should use *USE_SPATIAL*, *USE_DIRECTION*, *USE_ENERGY* or *USE_TIME* to tell to the source generator that our specific sampler requires a spatial, direction, energy and/or time sampler respectively. Previous values mean that the specified generic samplers will be called inside the specific sampler method *sample*. On the other hand, *USE_GENERIC* is used to tell to the source generator that the whole generic particle state (defined at *pen_particleState*) must be handled outside the specific sampler. Notice that, in this case, generic sampling will be done before the specific. Finally, *USE_NONE* specify that the sampling of the whole state is handled by the specific sampler without any generic sampler. The enumeration values can be combined using the — operator. For example, to delegate the spatial and directional sampling to generic samplers we will use something like lines shown at code 21.

```
mySpecificSampler() : abc_specificSampler<myParticleType>(USE_SPATIAL |
    USE_DIRECTION)
```
Code 21: Specific sampler with delegation on spatial and directional sampling.

At the above code, *mySpecificSampler* is our new specific sampler class, and *myParticleType* the compatible particle state type.

Finally, to include a specific sampler in PenRed, we must use the following macro,

*REGISTER_SPECIFIC_SAMPLER(Class, State, ID)*

instead of the used on generic samplers (*REGISTER_SAMPLER*). At this macro, *Class* is the name of our specific sampler class, *State* the name of the compatible particle state and *ID* the text identifier associated to our specific sampler. Finally, our header and source codes must be included at *src/particleGen/specific/includes/specificSamplers.hh* and *src/-particleGen/specific/source/specificSamplers.cpp* files respectively.

### 4.4.6 Generic state generators

#### src/particleGen

Previous sections have been described how particle states are sampled at PenRed. However, this samplers are not used directly at our main program. Instead, PenRed provides auxiliary classes to group and handle the different types of state samplers. The first one is the *pen_genericStateGen* class, which only handles generic state samplers, i.e. position, direction, energy and time samplers. This class contains the structure where generic samplers are registered via the provided macros. This structure consists of a set of *instantiator* classes (see section 4.3), one for each generic sampler type.

If someone explores the code, he/she will notice that this register variables are declared as functions and not as variables. The reason for this approach is that they must be declared as static members because we want to perform the registration for all *pen_genericStateGen* instances. In addition, this registration is done before the *main* function begins its execution. So, to avoid the *static initialization fiasco*, we must create the register instances via a static variable created inside a function. This function is a simple wrapper that always returns a reference to this object. One of this functions is defined at code 22.

```
instantiator<abc_spatialSampler>& pen_genericStateGen::spatialSamplers(){
  static instantiator<abc_spatialSampler>* ans =
                     new instantiator<abc_spatialSampler>();
  return *ans;
}
```
Code 22: Instantiate function for spatial samplers register

*pen_genericStateGen* class requires several components to complete the sampling. For example, requires a pointer to a *wrapper_geometry* class to call its *locate* function after the call to spatial sampling. The method *locate* will assign to *IBODY* and *MAT* the corresponding indexes to particle sampled position.

Obviously, *pen_genericStateGen* stores a pointer to each kind of generic samplers. This pointers will store some of the previously registered samplers. In addition, this class contains two member variables to control in which body or material are the particle sources (*sourceBody* and *sourceMat*). If *sourceBody* is greater or equal to 0 and/or *sourceMat* is greater than 0, the sampling process will discard all positions which are not located at the corresponding body or material index. Only one of the two variables is taken into account to discard and accept sampled particles. So, if both *sourceMat* and *sourceBody* takes a valid value to filter the sampled positions, only the *sourceBody* will be taken into account. Now, lets turn on *pen_genericStateGen* member methods.

- **samplersList**: Returns a standard C++ string with a formatted list of all registered samplers. Also, if takes a std::vector<std::string> for each sampler type as argument, fills each one with the names of each kind of sampler.
- **setGeometry**: Sets the geometry pointer to specified one.
- **Add sampler functions**: *pen_genericStateGen* contains a set of template functions to register new samplers. One example for spatial samplers is shown at code 23.

```
        template <class subclass>
        static int addSpatialSampler(const char* typeID){
            return spatialSamplers().addSubType<subclass>(typeID);
        }

```

Code 23: *pen_genericStateGen* add spatial sampler function

  Analogous functions have been implemented for each sampler type with the names *addSpatialSampler*, *addDirectionSampler*, *addEnergySampler* and *addTimeSampler* for spatial, direction, energy and time samplers respectively. These functions are used at the register macros.

- **Select sampler functions**: This set of functions, allows to select one of the registered samplers to be used. One select function exists for each sampler type with the names *selectSpatialSampler*, *selectDirectionSampler*, *selectEnergySampler* and *selectTimeSampler*. These functions try to instantiate the sampler specified by the introduced name and, then, call its configuration function. If both, instantiation and configuration, have been executed correctly, the pointer of the respective sampler type will be set to the instantiated sampler and a 0 value will be returned.
- **sample**: Taking a basic particle state (*pen_particleState*) as argument, the function will run the previously selected samplers to generate a new state. First, if some time sampler has been selected, particle initial age will be sampled. Then, spatial sampling and geometry locate method will be executed until the particle body and material indexes have the specified value by *sourceBody* or *sourceMat* respectively. Finally, the direction and energy will be sampled. These steps are implemented at the *sample* function, as we can see at the code 24. Notice that, for generic state samplers, each sample call is interpreted as a new history.

```
void pen_genericStateGen::sample(pen_particleState& state,
                                 pen_rand& random) const{

  //Reset state
  state.reset();

  //Perform time sampling if exists
  if(timeSampler != nullptr){
    timeSampler->sample(state,random);
  } else {
    state.PAGE = 0.0;
  }

  state.LAGE=LAGE;

  //Perform direction sampling
  directionSampler->sample(state,random);

  //Perform spatial sampling and locate the particle
  spatialSampler->sample(state,random);
  geometry->locate(state);

  //Check if source is restricted to specified body
  //or material
  if(sourceBody >= 0){
    while(state.IBODY != (unsigned)sourceBody){
      spatialSampler->sample(state,random);
      geometry->locate(state);
    }
  }
  else if(sourceMat > 0){
    while(state.MAT != sourceMat){
      spatialSampler->sample(state,random);
      geometry->locate(state);
    }
  }

  //Perform energy sampling
  energySampler->sample(state,random);

  //Its a primary (source) particle, set ILB[0] = 1
  state.ILB[0] = 1;
}
```

Code 24: Generic state generator sampling.

- **clear**: Delete selected samplers and set geometry pointer to *nullptr*.

This class has been described for the sake of completeness, however, notice that its modification is not advisable nor necessary to extend the PenRed functionality.

### 4.4.7 Specific state generators

#### src/particleGen

Specific state generators are a generalization of the generic state generators. These samplers use a common interface defined at *pen_specificStateGen* which consists of a template class with a single argument that specifies the compatible particle state type (*particleState*). This class family handles the particle state sampling allowing, but not requiring, the use of specific samplers.

As specific samplers can delegate partially or totally the generic sampling of the particle state (section 4.4.5), this generator type includes a generic state generator (section 4.4.6) as member variable to allow the use of generic samplings for position, direction, energy and time if required.

Regarding class methods, these are analogous to the discussed for *pen_genericStateGen*, but includes a new *add* and *select* method for specific samplers (*addSpecificSampler* and *selectSpecificSampler*). However, take into account that the *sample* method is slightly different, since it handles both, generic and specific samplers. The code of the *sample* method is shown at the code 25.

```
void sample(particleState& state,
            pen_KPAR& genKpar,
            unsigned long& dhist,
            const unsigned thread,
          pen_rand& random){

  genKpar = kpar;

  //Check if generic sampling is required
  if(useGeneric){
    //Perform generic sampling
    dhist = 1;
    genericGen.sample(state,random);

    //If specified, perform specific sampling
    if(useSpecific){
        specificSamplerVect[thread]->
            sample(state,genKpar,dhist,random);
    }
  }
  else{ //No generic sampling
    //If specified, perform specific sampling
    if(useSpecific){
        specificSamplerVect[thread]->
            sample(state,genKpar,dhist,random);

        //Locate particle in geometry
        geometry->locate(state);
    }
    else{
        //No sampler specified!
        printf("pen_specificStateGen:sample: Error: No generic nor specific sampler specified!\n");
        state.reset();
        genKpar = ALWAYS_AT_END;
        dhist = 0;
    }
  }
}
```

Code 25: Specific state generator sample method.

As we can see, the first differences appear at the function arguments. This *sample* method requires not only the particle state and a random number generation. This case has an extra input parameter (*thread*) and two output variables (*genKpar* and *dhist*). The extra input argument *thread* specify what thread is calling the sampling method. Remember that specific generators are not thread-save by construction and this property must be ensured on its implementation. Thus, a knowledge of the thread identifier could be useful to satisfy the thread-save condition.

On the other hand, the output variables *genKpar* and *dhist*, specify the particle type of the generated state and the history increment respectively. On generic samplers, the particle state is the same for all sampled states, which is specified by the *kpar* variable. Here, *kpar* stores a reference to the generic sampler *kpar*, and is the default value for *genKpar*. This could be changed if some specific sample has been specified.

At the start of the *sample* method, we check if the sampling method requires a generic sampling via the variable *useGeneric*. If the condition is accomplished, the generic part will be sampled using the generic sampler (*genericGen*) method *sample* (code 24). Also, *dhist* is set to 1. This variable and *genKpar* could be changed at the next step if some specific sampler is used i.e. if the *useSpecific* variable stores a *true* value. Notice that we use a vector of specific samplers and only the located at the position *thread* is used. This is because *pen_specificStateGen* creates an independent specific sampler for each thread to simplify the multi-thread handling.

Finally, if *useGeneric* stores a *false* as value, the only remaining option is to run a specific sampler. Notice that our specific sampler requires to handle different situations depending on its generic sampler requirements. This cases are summarised following,

- **Uses generic sampling**: No specific sampler has been provided. All the sampling is handled by generic samplers.
- **Uses generic and specific samplings**: In this case, is expected that the specific sampler only handles some specific characteristics of the particle type, but not the generic part. This includes the particle localization in the material system (*IBODY* and *MAT*). All these aspects will be handled by the generic sampler. However, specific sampler could manage the *dhist* and *genKpar*.
- **Uses specific sampling**: Specific sampler must handle all the state variables, the history increment *dhist* and particle type of the sampled state. In this case, the *locate* method of the geometry will be called after the specific sampler sampling function call.

## 4.5 Tallies

### src/tallies

At this section we will describe how tallies work. Tallies are used to extract information from simulations, such as energy/dose deposition, particle fluence, etc. Tallies use a common interface defined by the *pen_genericTally* class. This one has a set of virtual functions which should be called at different simulation points. Defining this functions, tallies can extract the required information from the simulation.

To explain that functions, suppose that we want to create a tally which objective is to print what are happening on each step of the simulation. We will name this tally *pen_tallyDummyLog*. Also, we must take into account when we consider that a particle begins its simulation. Any particle sampled or retrieved from a secondary particle stack, begins its simulation the first time it has been located at the geometry system into a non void region. That is, if a sampled particle state location is inside a void region ($MAT = 0$), the simulation of that particle is not considered as begun until that particle moves to a non void region of the geometry. If its direction doesn't aim to a non void region, this particle will escape and its simulation will never begins. Notice that all particles in the secondary stack should be created by a mother particle during their corresponding simulations. So, particles in that stacks should be located in a non void region. Thus, their simulations always will begin.

Notice also that a *particle simulation* is not the same as a *history simulation*, a *source simulation* or the *global simulation*. *History simulations* include all the *particle simulations* produced by the same primary particle. As well, *source simulations* include all history simulations produced by the source. Finally, the *global simulation* includes all *source simulations*.

Once we take care about these definitions, and before to discuss the available tally functions, it is advisable to talk about some arguments that are common for most of these functions. These are listed at the code 26,

```
    const double nhist
    const unsigned kdet
    const pen_KPAR kpar
    const pen_particleState& state
```

Code 26: Common variables for tally functions.

where *nhist* stores the actual history number, *kdet* the detector where the particle is located, *kpar* the corresponding particle index and, finally, *state* the actual particle state. Not all functions get all these parameters as arguments, furthermore, some ones requires another parameters. Well, let's finally discuss the available functions to define any tally via our dummy logger tally,

- **tally_beginSim**: This function doesn't get any argument and is called when the global simulation begins.

```
void pen_tallyDummyLog::tally_beginSim(){
    //This function is called at the
    //beginning of the global simulation

    printf("Global simulation begins\n");
}

```

Code 27: Dummy logger *tally_beginSim* function.

- **tally_endSim**: Called when the global simulation ends. This function takes the number of the last simulated history as argument.

```
void pen_tallyDummyLog::tally_endSim(const double nhist){

    //This function is called when the simulation ends,
    //"nhist" tells us the last history simulated

    printf("Simulation ends at history number %.0f\n",nhist);
}

```

Code 28: Dummy logger *tally_endSim* function.

- **tally_beginHist**: Called when a new history begins. When this function is called, the particle state must be already sampled. Take care that sampled particle can be created at void volume ($MAT = 0$) and, in this case, *step* will be called after to try to move the particle to the next non void volume. After that step call, **tally_move2geo** must be called, as we will discuss after.

```
void pen_tallyDummyLog::tally_beginHist(const double nhist,
                    const unsigned /*kdet*/,
                    const pen_KPAR /*kpar*/,
                    const pen_particleState& /*state*/){

    //This function is called when a new history begins
    printf("Simulation of history %.0f begins\n",nhist);
}

```

Code 29: Dummy logger *tally_beginHist* function.

- **tally_endHist**: Called when a history ends its simulation, i.e. primary particle and all his secondary generated particles has been simulated.

```
void pen_tallyDummyLog::tally_endHist(const double nhist){

  //This function is called when a history ends its simulation

  printf("Simulation of history %.0f ends\n",nhist);

}

```

Code 30: Dummy logger *tally_endHist* function.

- **tally_beginPart**: Called when a particle simulation begins.

```
void pen_tallyDummyLog::tally_beginPart(const double nhist,
              const unsigned /*kdet*/,
              const pen_KPAR kpar,
              const pen_particleState& state){

  //Called when the simulation of some particle beggins.
  //At this point, the particle must be inside the
  //geometry system into a non void region.

  printf("A %s from histoy %.0f begins its simulation.\n",
   particleName(kpar),nhist);
  printf("Is located at body %u wich material is %u.\n",
   state.IBODY,state.MAT);

}

```

Code 31: Dummy logger *tally_beginPart* function.

- **tally_endPart**: Called when a particle simulation ends.

```
void pen_tallyDummyLog::tally_endPart(const double nhist,
                const pen_KPAR kpar,
                const pen_particleState& /*state*/){
  //This function is called when the
  //simulation of a particle ends

  printf("A %s from histoy %.0f ends its simulation.\n",
   particleName(kpar),nhist);
}

```

Code 32: Dummy logger *tally_endPart* function.

- **tally_move2geo**: Called when a particle has been created (sampled by sources) into void region and has been moved forward to check if it arrives to any non void volume. If the particle reaches a non void volume, the simulation will continue and is supposed that *tally_beginPart* will be triggered. If not, *tally_endPart* will be called immediately without a *tally_beginPart* call. Take into account that this function is called when the particle simulation has not been started.

  Regarding the two extra variables (*dsef* and *dstot*) these provides the corresponding output values of the *step* function. As particle is initially on void region, *dsef* and *dstot* stores the same value. However, PenRed provides both values to allow future features.

```
void pen_tallyDummyLog::tally_move2geo(const double /*nhist*/,
                 const unsigned /*kdet*/,
                 const pen_KPAR kpar,
                 const pen_particleState& state,
                 const double dsef,
                 const double /*dstot*/){
  //This function is called when a new particle
  //is sampled into a void zone and, then, is
  //moved to find a geometry non void zone.

  printf("A %s has been created in a void region.",particleName(kpar));
  if(state.MAT != 0){
    printf("After moving it %E cm, has striked the geometry "
     "body %u which material is %u.\n"
     ,dsef,state.IBODY,state.MAT);
  }
  else{
    printf("There aren't any non void region on its"
     "direction (%E,%E,%E).\n",state.U,state.V,state.W);
    printf("The 'tally_beginPart' function will not be"
    "triggered for this particle,"
    "but the 'tally_endPart'.\n");
  }
}

```

Code 33: Dummy logger *tally_move2geo* function.

- **tally_localEdep**: Called when any particle losses energy locally during its simulation. For example, on interactions or particle absorption, but doesn't include energy losses by soft interactions during the step. Notice that part of this energy lost can be used to create new particles. So, energy lost is not the energy absorbed by the material. The argument *dE*, as its name suggest, stores the amount of deposited energy in eV.

```
void pen_tallyDummyLog::tally_localEdep(
                        const double /*nhist*/,
                        const pen_KPAR kpar,
                        const pen_particleState& /*state*/,
                        const double dE){

  //Called when the particle losses energy locally during its
  //simulation i.e. the energy loss is not continuous on a
  //traveled step

  printf("%s losses energy (%E eV) locally\n",
  particleName(kpar),dE);
}

```

Code 34: Dummy logger *tally_localEdep* function.

- **tally_step**: Called after step call during particle simulation. Is supposed that this function will be called after the update of particle age, i.e. after *dpage* call. This function will not be triggered when step is used to move a new sampled particle to the geometry when has been created at a void volume (as **tally_move2geo**), because the simulation has not already started.

  As extra argument, this function receives *stepData*. This is a *tally_StepData* structure which definition is shown at the code 35.

```
struct tally_StepData{
  double dsef;
  double dstot;
  double softDE;
  double softX;
  double softY;
  double softZ;
  unsigned originIBODY;
  unsigned originMAT;
  };

```

Code 35: *tally_StepData* structure.

First both variables (*dsef* and *dstot*) store the corresponding output values of the step function. Then, *softDE* stores the energy deposited due the traveled step. Previous energy should be considered to have been deposited at the point (*softX*,*softY*,*softZ*). Finally, *originIBODY* and *originMAT* save the particle *IBODY* and *MAT* values before the step respectively.

```
void pen_tallyDummyLog::tally_step(const double /*nhist*/,
                                   const pen_KPAR kpar,
                                   const pen_particleState& /*state*/,
                                   const tally_StepData& stepData){

  //Called after a step call during the particle
  //simulations i.e. after "tally_beginPart" has been
  //called for current particle

  if(stepData.dstot > 0.0)
    printf("%s moves %E cm in the origin material (%d) and "
     "%E cm in void regions.\n",
     particleName(kpar),stepData.dsef,
     stepData.originMAT,stepData.dstot);
  else
    printf("%s moves %E cm in the origin material (%d).\n",
     particleName(kpar),stepData.dsef,
     stepData.originMAT);

  //Check if the particle losses energy during the step
  if(stepData.softDE > 0.0){
    printf("During its travel, the particle losses %E eV.\n",stepData.
   softDE);
    printf("Considede that this energy has been "
    "deposited at P=(%E,%E,%E).\n",
  stepData.softX,stepData.softY,stepData.softZ);
  }
}

```

Code 36: Dummy logger *tally_step* function.

- **tally_interfCross**: Called when a particle cross an interface during simulation. An interface has been crossed when the step method returns a *NCROSS* with a non zero value. Moving sampled particles to geometry will not trigger this function.

```
void pen_tallyDummyLog::tally_interfCross(
                                   const double /*nhist*/,
                                   const unsigned /*kdet*/,
                                   const pen_KPAR kpar,
                                   const pen_particleState& /*state*/){
  //Called when the particle crosses
  //an interface during the simulation.
  printf("%s crossed an interface.\n",particleName(kpar));

}

```

Code 37: Dummy logger *tally_interfCross* function.

- **tally_matChange**: Called only when the particle change material during simulation. Moving primary particles to geometry will not trigger this function.

```
void pen_tallyDummyLog::tally_matChange(
                        const double /*nhist*/,
                        const pen_KPAR kpar,
                        const pen_particleState& state,
                        const unsigned prevMat){
  //Called when the particle crosses an interface and enters
  //in a new material during the simulation

  printf("%s go from material %u to material %u.\n",
   particleName(kpar),prevMat,state.MAT);

}

```

Code 38: Dummy logger *tally_matChange* function.

- **tally_jump**: Called immediately after jump call during the particle simulation.

```
void pen_tallyDummyLog::tally_jump(const double /*nhist*/,
                                   const pen_KPAR kpar,
                                   const pen_particleState& state,
                                   const double ds){
  //Called after jump function during the particle simulation
  printf("%s will try to travel %E cm following "
    "the direction (%E,%E,%E).\n",
    particleName(kpar),ds,state.U,state.V,state.W);
}

```

Code 39: Dummy logger *tally_jump* function.

- **tally_knock**: Called immediately after knock call during particle simulation.

```
void pen_tallyDummyLog::tally_knock(const double /*nhist*/,
                        const pen_KPAR kpar,
                        const pen_particleState& /*state*/,
                        const int icol){
  //Called after knock function during the simulation
  printf("%s has interact with the material via the"
   " interaction with index %d.\n",
   particleName(kpar),icol);
}

```

Code 40: Dummy logger *tally_knock* function.

- **tally_lastHist**: Called to update the number of previous histories. Tallies that use information about last registered history, such as phase space file, can't work property without this information.

```
void pen_tallyDummyLog::tally_lastHist(const double lasthist){
  //Called when a source begins its simulation.
  //"lasthist" tells us what is the initial history
  //to be simulated at this source.

  printf("New source begins at history number %.0f\n",
  lasthist);

}
```

```

```
Code 41: Dummy logger *tally_lastHist* function.

Despite the amount of available tally functions, a single tally doesn't requires to implement all of them. Only the necessary functions to get the interest data should be implemented. To specify which functions will be used by a tally, we use a similar method to the used at specific samplers. This consists of a set of flags passed as argument of the interface class *pen_genericTally* constructor. This flags are,

USE_BEGINSIM

USE_ENDSIM

USE_BEGINHIST

USE_ENDHIST

USE_MOVE2GEO

USE_BEGINPART

USE_ENDPART

USE_JUMP

USE_STEP

USE_INTERFCROSS

USE_MATCHANGE

USE_KNOCK

USE_ELOSS

USE_ANNIHILATION

USE_LASTHIST

where each flag name specify the corresponding function. Following we show an example of how to select the required functions for our implementation of the energy deposition at material tally (*pen_EdepMat* class),

```

pen_EdepMat() : pen_genericTally( USE_ELOSS |
                                   USE_BEGINPART |
                                   USE_BEGINHIST |
                                   USE_ENDHIST   |
                                   USE_MOVE2GEO)
{}
```
Code 42: Tally flag usage on implemented tallies.

In addition to optional tally functions, all tallies must implement some mandatory pure virtual methods. As our dummy logger doesn't extract any information, these function does nothing in its class. However, several examples can be found at PenRed built-in tallies. These methods are the following,

- **saveData**: Called when data report occurs. This function must store the data of interest in a file. Gets as argument the number of simulated histories. Don't care about the file name is needed, as the tally cluster will add some prefixes to all created files regarding its name, thread identifier and MPI process number.

- **flush**: This function handles, if needed, the calculation of final results. Commonly, tallies use auxiliary and temporal data buffers to avoid recalculating unchanged bins. After the flush call, is expected that all the actual measured data are stored at the corresponding final buffers. This function is called automatically before functions which requires the updated data until actual simulation point, like the *saveData* function.

- **sumTally**: This function gets a second tally instance of the same type to sum their results. The sum will be stored at the tally which calls *sumTally*. On success, this function returns 0. This function is used, for example, to sum-up the results of different threads on multi-threading simulations.

- **configure**: As other components, tallies require a configure initialization function. This one takes the usual arguments (a configuration structure and a verbose level) plus a geometry pointer and the material information, as we can see at the code 43.

```

virtual int configure(
        const wrapper_geometry& /*geometry*/,
        const abc_material* const /*materials*/[constants::MAXMAT],
        const pen_parserSection& /*config*/,
        const unsigned /*verbose*/) = 0;
```

Code 43: Tally configuration function.

As previous components, *config* provides all the specified parameters by the user to configure the tally inside a *pen_parserSection* structure (see section 3). Also, *verbose* specify the output verbose level.

On the other hand, *geometry* and *materials* provide all the simulation geometry and materials information which could be used by the tally. For example, a tally could be specific for some geometry or use that information to obtain the mass of a virtual mesh elements to calculate the absorbed dose.

In addition, the configuration function must register the necessary data to store the tally state in the member variable *dump*, which is provided by the tally interface. This *dump*, will handle the binary writing and reading of tally state. Both processes are automatically handled by the tally cluster, however the user must specify what variables must be dumped. To exemplify this simple process, see the code 44.

```

  //Register data to dump
  dump.toDump(edptmp,nmat);
  dump.toDump(edep,nmat);
  dump.toDump(edep2,nmat);
  dump.toDump(&nmat,1);
```

Code 44: Tally *dump* registration for *tallyEnergyDepositionMat*.

As we can see, we only requires one function to register the tally data. This function is *toDump*, and will register the data for both purposes, write and read binary dumps. *toDump*, gets two arguments. The first one, is a pointer to the data to register. The second one, is the number of elements in the array to register. Notice that *toDump* is overloaded to accept the basic data types, which simplify its usage.

Finally, implemented tallies must be registered using the *DECLARE_TALLY* and *REGISTER_TALLY* macros in tally header and source files respectively. This step is analogous to the samplers registration, and several examples can be found at PenRed built-in tallies.

Registered tallies can, and should, be used through *pen_tallyCluster* class, which manage tallies functions calls. Also, tallies should be created within *pen_tallyCluster* calling the method *createTally*, which signature is shown at code 45.

```cpp

int createTally(const char* ID,
        const char* name,
        const wrapper_geometry& geometry,
        const abc_material* const materials[constants::MAXMAT],
        const pen_parserSection& config,
        const unsigned verbose);
```

Code 45: Tally cluster *createTally* function.

Most variables are the required to configure the tally, as we had seen before. The additional parameters *ID* and *name* specify what type of tally are we trying to create and the assigned name respectively. The specified name will be used to construct a prefix for all files created inside the tally.

Once *createTally* has been called, the tally cluster will try to instantiate the specified tally and configure it. If the instantiation or configuration steps fails, a non zero value will be returned. If the tally creation has been executed successfully, their functions will be called automatically to extract simulation information.

This approach allows to incorporate new tallies with no changes in already implemented PenRed source files.

## 5 Constants, parameters and definitions

This section summarises some important constants and values of PenRed engine.

### 5.1 Important definitions

- **Void region/volume**: Geometry region with material index 0.
- **Primary particle**: Particle produced directly by a source and not as a consequence of any interaction of some other particle.
- **Secondary particle**: Particle produced by some interaction of other particle.
- **History**: A *history* consists of a primary particle and all its derived secondary particles.
- **Particle simulation**: Encompasses since the particle enters into a non void region of the geometry system, until it is absorbed or escaped from the system.
- **History simulation**: Includes all the particle simulations of the particles conforming the history.
- **Source simulation**: Includes all the history simulations due primary particles produced by the source.
- **Global simulation**: Consists of all source simulations.
- **Particle state**: All variables stored at the corresponding state structure, which base type should be the provided *pen_particleState*.

- **Object state**: We refer as a object state to the values of its internal variables, regardless if they are private, public or protected. Also, inherited variables are considered as part of the object state. For example, lets see the objects defined at the code 46. The state of an object of the first type *A*, consists of the variables *a1*, *a2* and *a3*. On the other hand, the state of an object *B* consists of *a1*, *a2*, *a3*, *b1* and *b2*, due the inheritance from the class *A*.

```

class A{
    protected:
        int a1,a2;
    public:
        double a3;

    A() : a1(0),a2(2),a3(5.3){}

    void do1();
    void do2();
};

class B : public A{
    private:
        int b1;
    public:
        double b2;

    B() : b1(0),{}

    void doB1();
    void doB2();
};

```

Code 46: Source configuration parameters

- **Verbose levels**: Verbose levels consists of a set of non negative indexes that indicates how verbose will be our execution. The meaning of that levels are summarised following,

  - 0: Any message should be printed.
  - 1: Only error messages should be printed.
  - 2: Errors, warnings and important information should be printed.
  - $> 2$: Prints all relevant information.

  Optionally, greater verbose levels could be used to filter very verbose information.

- **ILB values**: ILB labels follows the definition of the PENELOPE package, which are summarised at the table extracted from the PENELOPE manual [2]. Notice that our index begins from 0 because C++ syntax but at the FORTRAN code the first index is 1.

| ILB | description |
|---|---|
| ILB[0] | Generation of the particle; 1 for primary particles, 2 for their direct descendants and so on. Primary particles are assumed to be labelled with $ILB[0] = 1$. |
| ILB[1] | Parent particle *kpar* index (see table 3), only if $ILB[0] > 1$. |
| ILB[2] | Interaction mechanism $ICOL$ (see tables at section 5.4) that originated the particle, only when $ILB[0] > 1$. |
| ILB[3] | A non-zero value identifies particles emitted from atomic relaxation events and describes the atomic transition where the particle was released. The numerical value is, $$ILB[3] = Z \times 10^6 + IS1 \times 10^4 + IS2 \times 100 + IS3 \quad (7)$$ where $Z$ is the atomic number of the emitting atom and $IS1$, $IS2$ and $IS3$ are the labels of the active atomic electron shells (see table 7). For instance, $ILB[3] = 29010300$ designates a $K - L2$ x ray from copper ($Z = 29$), and $ILB[3] = 29010304$ indicates a $K - L2 - L3$ Auger electron from the same element. When $ILB[3] \neq 0$, the value of $ILB[2]$ indicates the itneraction mechanism that caused the initial vacancy in the decaying atom. |
| ILB[4] | This label can be defined by the user and must be transferred to all particle descendants. |

Table 1: Description of the $ILB$ components

## 5.2 Units

PenRed supposes the use of specific units internally. These units are summarised at the table 2.

| Magnitude | Unit |
|---|---|
| Length | cm |
| Energy | eV |
| Time | s |
| Material mass | g |
| Density | g/cm$^3$ |

Table 2: PenRed internal units.

## 5.3 Particle indexes

As we saw at section 4.1.7.1, each particle requires a index identifier provided via the enumeration *pen_KPAR*. Actually PenRed particle indexes and names are summarised at table 3.

| particle name | Enumeration identifier | Numerical index |
|---|---|---|
| electron | PEN_ELECTRON | 0 |
| gamma | PEN_PHOTON | 1 |
| positron | PEN_POSITRON | 2 |

Table 3: Particle indexes and names.

## 5.4 Particle interaction indexes

PenRed uses enumerations to indexing the interactions of each particle. This section show the corresponding index for each particle interaction.

| Interaction | Enumeration identifier | Numerical index |
|---|---|---|
| Elastic collision | BETAe_HARD_ELASTIC | 0 |
| Inelastic collision | BETAe_HARD_INELASTIC | 1 |
| Bremsstrahlung | BETAe_HARD_BREMSSTRAHLUNG | 2 |
| Inner shell interaction | BETAe_HARD_INNER_SHELL | 3 |
| Delta interaction | BETAe_DELTA | 4 |
| Soft interaction | BETAe_SOFT_INTERACTION | 5 |

Table 4: Electron interaction indexes.

| Interaction | Enumeration identifier | Numerical index |
|---|---|---|
| Rayleigh | GAMMA_RAYLEIGH | 0 |
| Compton | GAMMA_COMPTON | 1 |
| Photoelectric | GAMMA_PHOTOELECTRIC | 2 |
| Pair production | GAMMA_PAIR_PRODUCTION | 3 |
| Delta interaction | GAMMA_DELTA | 4 |

Table 5: Photon interaction indexes.

| Interaction | Enumeration identifier | Numerical index |
|---|---|---|
| Elastic collision | BETAp_HARD_ELASTIC | 0 |
| Inelastic collision | BETAp_HARD_INELASTIC | 1 |
| Bremsstrahlung | BETAp_HARD_BREMSSTRAHLUNG | 2 |
| Inner shell interaction | BETAp_HARD_INNER_SHELL | 3 |
| Annihilation | BETAp_ANNIHILATION | 4 |
| Delta interaction | BETAp_DELTA | 5 |
| Soft interaction | BETAp_SOFT_INTERACTION | 6 |

Table 6: Positron interaction indexes.

## 5.5 Atomic electron shells

The atomic electron shells labels are designed following the indexes used at PENELOPE code. These indexes are summarised at the table 7, which has been extracted from the PENELOPE manual [2].

| Label | Shell | Label | Shell | Label | Shell |
|---|---|---|---|---|---|
| 1 | K ($1s_{1/2}$) | 11 | N2 ($4p_{1/2}$) | 21 | O5 ($5d_{5/2}$) |
| 2 | L1 ($2s_{1/2}$) | 12 | N3 ($4p_{3/2}$) | 22 | O6 ($5f_{5/2}$) |
| 3 | L2 ($2p_{1/2}$) | 13 | N4 ($4d_{3/2}$) | 23 | O7 ($5f_{7/2}$) |
| 4 | L3 ($2p_{3/2}$) | 14 | N5 ($4d_{5/2}$) | 24 | P1 ($6s_{1/2}$) |
| 5 | M1 ($3s_{1/2}$) | 15 | N6 ($4f_{5/2}$) | 25 | P2 ($6p_{1/2}$) |
| 6 | M2 ($3p_{1/2}$) | 16 | N7 ($4f_{7/2}$) | 26 | P3 ($6p_{3/2}$) |
| 7 | M3 ($3p_{3/2}$) | 17 | O1 ($5s_{1/2}$) | 27 | P4 ($6d_{3/2}$) |
| 8 | M4 ($3d_{3/2}$) | 18 | O2 ($5p_{1/2}$) | 28 | P5 ($6d_{5/2}$) |
| 9 | M5 ($3d_{5/2}$) | 19 | O3 ($5p_{3/2}$) | 29 | Q1 ($7s_{1/2}$) |
| 10 | N1 ($4s_{1/2}$) | 20 | O4 ($5d_{3/2}$) | 30 | outer shells |

Table 7: Atomic electron shells indexes.

# 6 Framework usage

This section, address how to use the main program provided with PenRed package. This one allows the user to make simulations without programming. Instead, the user will use a configuration file which structure will be explained below. Configuration files must follow the internal data library format explained on section 3. Next sections will explain how to configure the required components to any simulation, i.e. sources, geometry, materials, tallies and global parameters.

## 6.1 Material creation

Material files should be created using the PENELOPE database, which is provided within PenRed package at,

```
dataBases/penmaterials
```

In this folder, is stored both, the database inside the *pdfiles* folder and the code to compile the material builder. Last one is a literal translation to C++ of the original FORTRAN code used by PENELOPE to create the materials. Thus, outputs of both, C++ and FORTRAN, versions must be perfectly equivalent and usable as input for PenRed. The following line is an example to compile this code using the C++ GNU GCC compiler,

```
g++ -o createMat material.cpp
```

where *createMat* is the executable name. Once the code has been compiled, execute it at the same folder where the database *pdfiles* is and follow the program instructions to create the material.

## 6.2 Particle sources

To specify a particle source, the user must follow the pattern shown at code 47,

```
sources/type/name/parameter/path value
```

Code 47: Source configuration parameters

where *type* must be set to *generic* or *polarized*, whose are used to specify if the source samples generic particle states or gamma polarized states respectively. With field *name*, the user specifies a custom name for the particle source. Finally, *parameter/path* specifies the configuration parameter to set with the introduced *value*. For example, the code 48 shows the basic parameters used by all sources for a generic source with name *source1*.

```
sources/generic/source1/nhist 1.0e6          (mandatory)
sources/generic/source1/kpar "gamma"         (mandatory)
sources/generic/source1/record-time true     (optional)
sources/generic/source1/source-body 1        (optional)
sources/generic/source1/source-material 1    (optional)
```

Code 48: Source generic configuration parameters

That source will produce $10^6$ primary gamma particles and enable time recording. So the parameter *nhist*, *kpar* and *record-time* sets the number of particles to generate by the source, the particles type (electron, gamma or positron) and if time recording is enabled or disabled respectively.

In addition we can specify at what body or material the particles must be produced using the parameters *source-body* and *source-material* respectively. Both parameters expects an integer index, which specify the source body or material. However, notice that only one restriction, can be used at the same source. When a particle is sampled in a body or material different that the specified, the spatial sampling will be repeated until the condition is fulfilled.

Note that each parameter requires a specific type. *nhist* requires a number, *kpar* a text (using the double quotes) and *record-time* a boolean (true or false). Once we have the generic parameters, we need to specify the samplers to use i.e. spatial, direction, energy, time and specific. To specify what kind of sampler we are configuring, we will use the keywords *spatial*, *direction*, *energy*, *time* and *specific*. Each sampler has its own parameters, whose can include custom paths, but all requires a parameter named *type*, which sets the sampler to use. For example, to select a mono-energetic energy sampler on previous source, the line with the required type is shown at code 49.

```
sources/generic/source1/energy/type "MONOENERGETIC"
```

Code 49: Mono-energetic sampler configuration example

Mono-energetic sources only require a single parameter in addition to type, the particle energy. Next code set a sampling energy of 30 $keV$,

```
sources/generic/source1/energy/energy 3.0e7
```

Code 50: Mono-energetic sampler configuration example

The other samplers are configured analogously. An example of complete generic source is shown at code 51, where the lines that begins with # are considered comments and ignored by the parser.

```
#-------------------------
#  Source 1
##########################

sources/generic/source1/nhist 1.0e6
sources/generic/source1/kpar "gamma"
sources/generic/source1/record-time true

# Directional sampling
##########################

sources/generic/source1/direction/type "CONE"


```

```
# Set theta
sources/generic/source1/direction/theta 0.0

# Set phi
sources/generic/source1/direction/phi 0.0

# Set oberture (alpha)
sources/generic/source1/direction/alpha 5.0


# Energy sampling
##########################

sources/generic/source1/energy/type "MONOENERGETIC"

# Set energy
sources/generic/source1/energy/energy 3.0e7


# Spatial sampling
##########################

sources/generic/source1/spatial/type "POINT"

# Set particle origin
sources/generic/source1/spatial/position/x 0.0
sources/generic/source1/spatial/position/y 0.0
sources/generic/source1/spatial/position/z -25.0
```

Code 51: Complete source configuration example

A source without specific sampler like the previous one requires, at least, spatial, energy and direction sampler, being the time sampler optional. If a specific sampler is used, the required generic samplers depends on the specific sampler itself, and may need some or any generic sampler. At the examples folder we can find configuration files with different sources.

### 6.2.1 Spatial source samplers

#### 6.2.1.1 Point

The spatial point sampler needs the origin position of particles generation at the configuration file. These values are doubles that correspond to the $(x, y, z)$ point coordinates, in cm. Following there is an example of this kind of source sampling corresponding to the 1-disc example.

```
# Spatial sampling
##########################

sources/generic/source1/spatial/type "POINT"

# Set particle origin
sources/generic/source1/spatial/position/x 0.0
sources/generic/source1/spatial/position/y 0.0
sources/generic/source1/spatial/position/z -0.0001
```

Code 52: Spatial source POINT sampler

#### 6.2.1.2 Box

This spatial sampler creates a box where particles are generated with uniform probability. The configuration file contains the information of the box origin introducing the $(x, y, z)$ coordinates values, in cm. Moreover, it contains the size of the box in each axis: $(dx, dy, dz)$ in cm. Each box size must be, at least, zero. All these values must be doubles. An example of this part of the configuration file is shown below.

```
# Spatial sampling
##########################

sources/generic/source1/spatial/type "BOX"


# Set box size
sources/generic/source1/spatial/size/dx 0.75
sources/generic/source1/spatial/size/dy 0.75
sources/generic/source1/spatial/size/dz 1.25

# Set particle origin
sources/generic/source1/spatial/position/x 0.0
sources/generic/source1/spatial/position/y 0.0
sources/generic/source1/spatial/position/z -0.0001
```

Code 53: Spatial source BOX sampler

### 6.2.2 Direction samplers

#### 6.2.2.1 Cone

Primary particles generated by the source are sampled in a conical beam. In this case the oberture of the cone must be specified and the particles are sampled uniformly inside the solid angle. Polar (*theta*) and azimutal (*phi*) angles determine the solid angle with an aberture (*alpha*). Angular values are in degrees. The limits of the oberture define a monodireccional souce if $alpha = 0.0$ and an isotropic source if $alpha = 180.0$. All of this values must be a double.

The following lines show the directional sampling of the configuration file of example 2-plane.

```

# Directional sampling
##########################

sources/generic/source1/direction/type "CONE"


# Set theta
sources/generic/source1/direction/theta 0.0

# Set phi
sources/generic/source1/direction/phi 0.0

# Set oberture (alpha)
sources/generic/source1/direction/alpha 5.0
```

Code 54: Direction CONE sampler

#### 6.2.2.2 Sphere

In this case the direction of the particles generated are sampled in a section of a sphere. The information needed in this direction sampler are the direction in each axis $(u, v, w)$, the polar oberture introducing the minimum and maximum value of the polar angle $(theta0, theta1)$, and the azimutal oberture introducing the minimum value of phi angle $(phi0)$ and the oberture of phi angle $(dphi)$. All of this values must be doubles.

### 6.2.3 Energy samplers

#### 6.2.3.1 Monoenergetic

This energy sampler generates a monoenergetic beam of primary particles with the specified energy in eV. The *energy* value is introduced as a double, and must be at least zero. An example of its use in the configuration file is shown below. This lines correspond to the example 4-x-ray-tube, file tube.in.

```

# Energy sampling
###########################

sources/generic/source1/energy/type "MONOENERGETIC"

# Set energy
sources/generic/source1/energy/energy 1.5e5
```

Code 55: Energy samplers MONOENERGETIC

#### 6.2.3.2 Intervals

Interval samplers generates primary particles on an specified spectral line with the probability assigned for each line. In the configuration file, user writes the number of intervals *ninterval* (an interger) that they will use in the simulation. Then, the energy range of each interval, $[lowE, topE]$; low energies of the different intervals are arranged in an $lowE$ array and the same for top energies as we can see in the following configuration file. Finally, the probabilities of each interval are wrote in the *probabilities* array. The next code show the 3-detector example, file detector1.in. Energy range values and probabilities are doubles and must be, at least, zero.

```

# Energy sampling
###########################

sources/generic/source1/energy/type "INTERVALS"


# Set intervals
sources/generic/source1/energy/nintervals 2


# Set energies
sources/generic/source1/energy/lowE [1.17e6,1.33e6]
sources/generic/source1/energy/topE [1.17e6,1.33e6]

# Set probabilities
sources/generic/source1/energy/probabilities [50.0,50.0]
```

Code 56: Energy samplers INTERVALS

### 6.2.4 Time samplers

#### 6.2.4.1 Decay

This sampler generates, randomly, an exponential timing decay of events from a radioactive element. The time interval is introduced by the user with $[time0, time1]$ and negative values are not allowed. In addition, the $halfLife$ of the radioactive element is required.

## 6.3 Geometries

Each simulation only allows to use a single geometry. Configuration parameters use the prefix *geometry*. As sources, any geometry requires to specify its type, and the rest of parameters are specific of each geometry type.

### 6.3.1 Quadric

This geometry type is used at all the examples provided by PENELOPE that are reproduced at PenRed package. Code 57 shows the geometry configuration for the first simulation example (1-disc).

```
geometry/type "PEN_QUADRIC"
geometry/input-file "disc.geo"
geometry/processed-geo-file "report.geo"

geometry/dsmax/1 1.0e-4
geometry/kdet/1  1
```

Code 57: Complete geometry configuration example

That simulation uses a quadric base geometry, which require to specify the type and the *input-file* parameter with the path to the file where geometry has been defined. Other parameters *processed-geo-file*, *dsmax* and *kdet* are optional. First parameter, *processed-geo-file*, specify a file to generate a geometry report. On the other hand, *dsmax* specify the maximum distance allowed for electrons and positrons to jump and *kdet* the detector identifier for body with alias " 1". Optionally, it is possible to specify local energy absorption for any body and particle. At code 58 we shown an example which sets local energy absorption for each particle to 10 $keV$ for the body with alias " 2".

```
geometry/eabs/2/electron   1.0e4
geometry/eabs/2/gamma      1.0e4
geometry/eabs/2/positron   1.0e4
```

Code 58: Configuration of geometry absorption energy example

### 6.3.2 Voxel

Voxelized geometries consists of a 3D matrix of regular prisms elements with a specific material and density factor. The density factor is used to specify heterogeneities in the density of the materials. For instance, a voxel with a density factor of 1.10 will have a density 10% greater than the material nominal density (specified at the material file). Likewise, a voxel with a density factor of 0.8 is considered to have a density 20% lower than the nominal material density. To use the original material density in a voxel, this factor must be set to 1.

This kind of geometries requires to set $VOXEL$ as geometry type and provide the path to the file where geometry has been defined (*filename*). The files which stores the voxelized geometry, consists of a binary data dumped by the *pen_voxelGeo* class. One example of how to create a geometry file using that class can be found at the test code, *src/tests/geometry/voxels/read_Dump.cpp*

which creates a random filled voxel geometry, stores it to a file and, finally, load the file to compare with the original created geometry. Also, another example can be found at the utility *geo2voxel*, which source file is located at,

*src/utilities/geometry/geo2voxel.cpp*

This utility, instantiate a geometry, specified by the user, and creates a voxelized geometry file according to that geometry. For that, uses the *locate* method to map each voxel material and density.

Following with the configuration parameters, the user must specify $(nx, ny, nz)$ and $(dx, dy, dz)$ to describe the number of voxels and its dimensions, respectively, in each axis. Also, *dsmax* can be specified as we done for quadric geometries. This one is done following the template,

*../../dsmax/nvalue*

where $n$ selects the material where *dsmax* will be applied and *value* the *dsmax* value. An example of configuration file for voxelized geometries is shown at code 59.

```
geometry/type "VOXEL"
geometry/filename "3blocks.vx"
geometry/nvoxels/nx 60
geometry/nvoxels/ny 60
geometry/nvoxels/nz 80
geometry/voxel-size/dx 0.2
geometry/voxel-size/dy 0.2
geometry/voxel-size/dz 0.1

geometry/dsmax/1 0.05
geometry/dsmax/2 0.02
geometry/dsmax/3 0.02
geometry/dsmax/4 0.50
```

Code 59: Configuration of voxelized geometry example

### 6.3.3 DICOM

Medical images are usually stored using the international standard of Digital Imaging and Communications in Medicine (DICOM). PenRed implements a DICOM geometry module to convert the DICOM file to a voxel geometry automatically. This converted geometry consists of a 3D grid of regular parallelepipeds elements. With this module, PenRed is capable to run simulations directly using DICOM files. As other PenRed components, DICOM geometries require configuration structure to be used. This configuration includes the following parameters,

- **type**: Geometry type, must be set to "DICOM".
- **directory**: Specifies the path where the DICOM images are stored. Notice that all the DICOM images found at that folder are expected to be from the same image.
- **calibration**: This optional parameter is only used for CT images. This must be specified as an array of numbers which belong to a polynomial calibration to convert from Hounsfield Units (HU) to density ($g/cm^3$). If the calibration is not specified for

CT images, the raw data will be used and the density must be assigned using other techniques, as we will see below.

- **default/material**: Specifies default material index for all voxels which material has not been assigned by other methods.
- **default/density**: Specifies default density for all voxels which density has not been assigned by any of the available methods.
- **intensity-ranges**: Provides a subsection to assign a material index and density value to all voxels inside the specified pixel value ranges. This subsection consists of the following parameters,
  - **material**: Material index to assign.
  - **density**: Density value ($g/cm^3$) to assign.
  - **low**: Lower range pixel value to assign this material and density.
  - **top**: Upper range pixel value to assign this material and density.

  So, all voxels with intensity values in the range $[low, top)$ will be assigned with the material index *material* and the density value *density*. To differentiate between ranges, each one requires a unique name. This name must not be the material name, but is advisable for debug purposes. An example of this configuration is shown at code 60, where we define an interval named *Air*.

```
geometry/intensity-ranges/Air/material 1
geometry/intensity-ranges/Air/low -2000
geometry/intensity-ranges/Air/top -500
geometry/intensity-ranges/Air/density 0.001290
```

Code 60: DICOM intensity ranges configuration

- **contours**: Subsection that allows to the user to assign materials and densities according to contours stored inside the DICOM image. Each subsection of this type defines a single contour which name is specified at all the parameters paths. Notice that the contour name must coincide with the contour name stored at the corresponding DICOM file. Also, it contains the following parameters,
  - **material**: Material to assign to this contour.
  - **density**: Density to assign to this contour ($g/cm^3$).
  - **priority**: A priority value to control which contours are overwritten by other ones. Contour with lower priority values will be overwritten by highest priority values.

  For example, to configure a contour named *target*, the configuration file should contain something like the lines shown at code 61.

```
geometry/contours/target/material 1
geometry/contours/target/density 1.05
geometry/contours/target/priority 1.0
```

Code 61: DICOM contour configuration

- **ranges**: Subsection analogous to *intensity-ranges* but using density ranges instead of pixel values. Thus it could be used to specify the voxels material indexes via density ranges. Notice that, at this point, densities must be set using calibration information. This subsection consists of the following parameters,

- **material**: Material index to assign.
- **density-low**: Minimum density value for this range.
- **density-top**: Maximum density value for this range.
  All voxels with a density value between $(density - low, density - top]$ will be assigned with the material index *material*.
- **print-ASCII**: This true/false optional configuration parameter, can be set to *true* to print the processed DICOM voxel data in ASCII format to a file with the name *dicomASCII.rep*.

Notice that this geometry type presents several ways to assign material indexes and densities to voxels. Due to this characteristic, exists a method hierarchy where preferential methods overwrite the others. This preference is shown at figure 1.

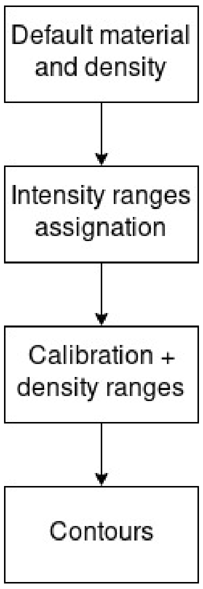
Default material and density
Intensity ranges assignation
Calibration + density ranges
Contours

Figure 1: DICOM voxels material an density assign methods hierarchy.

Notice that the allowed image modalities for DICOM geometry type are, by the moment, Computed Tomography (CT), Ultrasound (US), Radiotherapy Structure Set (RTSTRUCT) and Radio-therapy Plan (RTPLAN).

A DICOM example where voxel density and material are set using only intensity ranges can be found at the *example* folder. This one reproduces the GEANT IV DICOM simulation example using the DICOM image developed at [7]. So, the user must download that DICOM to reproduce the example.

## 6.4 Materials

First of all, notice that material files used for PenRed simulations are the same as the used by the original PENELOPE FORTRAN code. So, materials should be generated using the tools and data base provided at original PENELOPE package.

To specify a material configuration, the user must follow the pattern of code 62, where *materials* is a constant text, *material-name* is a text identifier for that material and, finally, *parameter/path* and *value* are the parameters and values to specify.

```
materials/material-name/parameter/path value
```

Code 62: Material configuration pattern

As an example of material configuration we will show the copper material configuration segment used at first example (code 63)

```
materials/cu/number 1

materials/cu/eabs_e- 1.0e3
materials/cu/eabs_e+ 1.0e3
materials/cu/eabs_gamma 1.0e3

materials/cu/C1 0.05
materials/cu/C2 0.05

materials/cu/WCC 1.0e3
materials/cu/WCR 1.0e3

materials/cu/filename "Cu.mat"
```

Code 63: Complete material configuration example

As we can see, the parameters to specify materials are analogous to the required by the PENELOPE FORTRAN version. The material number must be greater than 0, thus material 0 is considered as void.

## 6.5 Tallies

Configuration for tallies are similar to sources. The pattern to configure tallies is at code 64, where *tallies* is a constant text, *tally-name* is a text identifier for this tally, and *parameter/path* and *value* sets the tally parameters, whose depends on tally type. As sources, exists several tally types, so the user must specify the parameter *type* on each created tally.

```
tallies/tally-name/parameter/path value
```

Code 64: Tally configuration pattern

Code 65 shows an example to configure a cylindrical dose distribution tally, which requires limits for radial distance (*rmin* to *rmax*), number of radial bins *nbinsr*, limits for $z$ axis (*zmin* to *zmax*) and the number of $z$ bins (*nbinsz*). The optional parameter *print-xyz* serves to enable (*true*) or disable ($false$) more information about radial coordinates at the output file: the low and the average value of these coordinates.

```
tallies/cylDoseDistrib/type "CYLINDRICAL_DOSE_DISTRIB"
tallies/cylDoseDistrib/print-xyz true
tallies/cylDoseDistrib/rmin 0.0
tallies/cylDoseDistrib/rmax 30.0
tallies/cylDoseDistrib/nbinsr 60
tallies/cylDoseDistrib/zmin 0
tallies/cylDoseDistrib/zmax 30.0
tallies/cylDoseDistrib/nbinsz 60
```

Code 65: Tally configuration pattern

An aspect to consider when we are choosing the limits, is that the behaviour of the intervals is $[min, max)$. Thus *rmin* and *zmin* are out of their respective intervals.

The next subsections describe briefly the data measured in each tally and provide an example to use them at the configuration file.

### 6.5.1 Radial and Cylindrical Dose Distribution

This tally measures the absorbed dose in (eV/g) for each radial bin in the range of $[rmin, rmax)$, in cm. If user wants to measure depth absorbed dose, the tally measurement is in (eV cm/g) per history and depth bin is in the range of $[zmin, zmax)$ in cm. Below is shown the variables of this tally,

- **type "CYLINDRICAL_DOSE_DISTRIB"**: Type name of the tally.
- **print-xyz**: This variable can be set to **true** for more information. In this case, two values per bin coordinate are given, the low and the average value. For z coordinate the average is considered at the middle point of the bin. For the r coordinate the average is weighted with a weight proportional to the radius r. If $print - xyz$ is not specified, **false** value will be assigned.
- **rmin**: Minimum value of the radial coordinate. Must be greater than zero and lower than *rmax*.
- **rmax**: Maximum value of the radial coordinate.
- **nbinsr**: Number of radial bins. Must be at least 1.
- **zmin**: Minimum value of the depth coordinate. Must be lower than *zmax* when *nbinsz* is set greater to zero.
- **zmax**: Maximum value of the depth coordinate. Must be set to *zmin* when depth absorbed dose is not measured.
- **nbinsz**: Number of depth bins. Must be set to zero when depth absorbed dose is not measured, otherwise, must be at least 1.

The limits of radial and depth intervals must be doubles, while the number of bins in each case must be an integer. The output filename of this tally ends with *cylindricalDoseDistrib.dat*. In case of we don't want to measure depth absorbed dose, the output filename ends with *radialDoseDistrib.dat*.

An example of this tally in a configuration file is shown at code 66, whose lines correspond to the file *disc.in* of the example 1-disc.

```
tallies/cylDoseDistrib/type "CYLINDRICAL_DOSE_DISTRIB"
tallies/cylDoseDistrib/print-xyz true      (optional)
tallies/cylDoseDistrib/rmin 0.0            (mandatory)
tallies/cylDoseDistrib/rmax 0.01           (mandatory)
tallies/cylDoseDistrib/nbinsr 50           (mandatory)
tallies/cylDoseDistrib/zmin 0.0            (mandatory)
tallies/cylDoseDistrib/zmax 0.005          (mandatory)
tallies/cylDoseDistrib/nbinsz 100          (mandatory)
```

Code 66: Cylindrical dose distribution tally configuration

### 6.5.2 Emerging Particles Distribution

On the one hand, this tally measures the number of particles that left the geometry for each energy value in a energy range of $[emin, emax)$ expressed in particles/(eV history). We consider two cases in this tally. First, down bound emerging particles when the direction $W$ of them is lower or equal to zero. Then, up bound case if $W$ is greater than zero. On the other hand, the tally measures the number of particles that left the geometry for steradian determined by the theta and phi angular values, in degrees, of the deviation of the particle. In this case, measurements are in particles/(sr history).

Tally information needed in the configuration file is shown below,

- **type "EMERGING_PART_DISTRIB"**: Type name of the tally.
- **emin**: Minimum energy value. Must be lower than *emax*.
- **emax**: Maximum energy value.
- **nBinsE**: Number of energy bins. Must be at least 1.
- **nBinsTheta**: Number of polar bins. Must be greater than zero.
- **nBinsPhi**: Number of azimuthal bin, Must be greater than zero.

The energy limits must be doubles while the number of bins must be an integrer. The output files of this tally end as: *emergin-downbound.dat*, *emergin-upbound.dat*, *emergin-angle.dat*, with information for theta and phi values and *emergin-polar-angle.dat* with only angular information of theta values.

The code 67 is an example of this tally at the configuration file *detector1.in* of the example 3-detector-1:

```
tallies/EmergingPartDistrib/type "EMERGING_PART_DISTRIB"
tallies/EmergingPartDistrib/emin 0.0              (mandatory)
tallies/EmergingPartDistrib/emax 1.45e6           (mandatory)
tallies/EmergingPartDistrib/nBinsE 280            (mandatory)
tallies/EmergingPartDistrib/nBinsTheta 45         (mandatory)
tallies/EmergingPartDistrib/nBinsPhi 18           (mandatory)
```

Code 67: Emerging particles tally configuration

### 6.5.3 Energy Deposition Body

This tally measures the energy in eV deposited in each body per history.

Tally information of the variables needed in the configuration file is shown below,

- **type "EDEP_BODY"**: Type name of the tally.
- **nBody**: Number of bodies in which we calculate the deposited energy.

The *nBody* variable must be an integer. The output filename ends with *bodyEnergyDeposition.dat*.

An example of this tally in a configuration file is shown at code 68. It corresponds to the file *plane.in* of the example 2-plane.

```
tallies/bodyEDep/type "EDEP_BODY"
tallies/bodyEDep/nBody 2            (mandatory)
```

Code 68: Energy deposition in body tally configuration

### 6.5.4 Energy Deposition Material

This tally measures the energy in eV deposited in each material per history.

The variables of this tally needed in the configuration file are shown below,

- **type "EDEP_MAT"**: Type name of the tally.
- **nmat**: Number of materials in which user can calculate the deposited energy.

This *nmat* value must be an integer. The output file ends as *materialEnergyDeposition.dat*

Code 69 shown an example of the configuration file *detector1.in* belonging to the example 3-detector-1 where we can see an example of the tally information.

```
tallies/matEDep/type "EDEP_MAT"
tallies/matEDep/nmat 2            (mandatory)
```

Code 69: Energy deposition in material tally configuration

### 6.5.5 Impact Detector

This tally includes different measures: fluence spectrum, particle energy spectrum and particle age.

- Fluence: integrates the spectral fluence over the detector volume in cm/eV. The output file contains the fluence for each particle type and the total fluence in the specified detector. The filename ends with *fluenceTackLength-num.dat*. Where *num* is the number assigned to the detector. We can measure the fluence for different detector volumes.

- Energy spectrum: reports the energy spectrum of the particles that enter in a detector volume. Particles created inside the detector, i.e, secondary particles, are not considered in this tally. Units are expressed in 1/(eV history). The output file contains the energy spectrum for each particle type and the total spectrum in the specified detector is also measured. The filename ends with *spectrum-impdet-num.dat*, where *num* is the number assigned to the detector.

- Age: reports the age distribution of the particles that suffer an impact at the considered detector, in 1/(seconds history). The output file that ends with *age-impdet-num.dat*, contains the probability distribution along each time interval for the specific detector over all simulated particles. As in the previous cases, *num* is the number assigned to the detector.

- Energy deposition: reports the energy deposited spectrum at the considered detector. Units are expressed in 1/(eV history). The output file ends with *energyDeposition-impdet-num.dat*, where *num* is the number assigned to the detector. This file contains the probability distribution along each energy interval for the specific detector over all simulated particles.

Tally information needed in the configuration file:

- **type "IMPACT_DET"**: Type name of the tally.
- **detector**: Detector index on which measurements of this tally are taken.
- **fluence**: Must be set to **true** to obtain fluence measurements, otherwise set to **false**. If this information is not specified **false** value will be assigned.
- **emin**: Minimum energy value.
- **emax**: Maximum energy value. Must be greater than *emin.*
- **nbin-energy**: Number of energy bins. Used for fluence, energy spectrum and energy deposition. Must be at least 1.

- **linearScale-energy**:
- **spectrum**: Must be set to **true** to obtain spectrum measurements, otherwise set to **false**. If this information is not specified **false** value will be assigned.
- **linearScale-spc**: Determines if the output measurements are expressed in linear scale (**true**) or in logarithmic scale (**false**). If not scale is specified, linear scale will be set.
- **age**: Must be set to **true** to obtain age measurements, otherwise set to **false**. If this information is not specified **false** value will be assigned.
- **linearScale-age**: Determines if the output measurements are expressed in linear scale (**true**) or in logarithmic scale (**false**). If not scale is specified, linear scale will be set.
- **nbin-age**: Number of age interval bins. Must be at least 1.
- **age-min**: Minimum value of age interval.
- **age-max**: Maximum value of age interval. Must be greater than *age-min*.
- **enegy-dep**: Must be set to **true** to obtain energy deposition measurements, otherwise set to **false**. If this information is not specified **false** value will be assigned.
- **linearScale-edep**: Determines if the output measurements are expressed in linear scale (**true**) or in logarithmic scale (**false**). If not scale is specified, linear scale will be set.

The interval limits in each case must be doubles and the bin numbers must be an integer.

Next, there are two examples to show the different measures obtained with this tally. First, code 70 shows a part of the configuration file *plane.in* of the 2-plane example, where we can see fluence, spectrum and age information. If fluence, spectrum, age or energy deposition are activated, their corresponding fields are mandatory. Otherwise, this information is not required because this part of the tally is no measured and they are optional.

```
tallies/ImpactDetector/type "IMPACT_DET"
tallies/ImpactDetector/detector 1                (mandatory)
tallies/ImpactDetector/fluence true              (optional)
tallies/ImpactDetector/emin 1.0e5                (mandatory/optional)
tallies/ImpactDetector/emax 3.5e7                (mandatory/optional)
tallies/ImpactDetector/nbin-energy 100           (mandatory/optional)
tallies/ImpactDetector/linearScale-fln true      (optional)
tallies/ImpactDetector/spectrum true             (optional)
tallies/ImpactDetector/age true                  (optional)
tallies/ImpactDetector/linearScale-age false     (mandatory/optional)
tallies/ImpactDetector/nbin-age 100              (mandatory/optional)
tallies/ImpactDetector/age-min 1.0e-9            (mandatory/optional)
tallies/ImpactDetector/age-max 1.0e-8            (mandatory/optional)
```

Code 70: Impact detector tally configuration example 1

And then, code 71 shows some lines of the configuration file *detector1.in* that belongs to the example 3-detector-1, to see an example of energy deposition information.

```
tallies/ImpactDetector/type "IMPACT_DET"
tallies/ImpactDetector/detector 1                (mandatory)
tallies/ImpactDetector/emin 0.0e0                (mandatory/optional)
tallies/ImpactDetector/emax 1.45e6               (mandatory/optional)
tallies/ImpactDetector/nbin-energy 280           (mandatory/optional)
tallies/ImpactDetector/energy-dep true           (optional)
tallies/ImpactDetector/linearScale-edep true     (optional)
```

Code 71: Impact detector tally configuration example 2

### 6.5.6 Spatial Dose Distribution

This tally measures the 3D absorbed dose distribution along the intervals $[xmin, xmax)$, $[ymin, ymax)$ , $[zmin, zmax)$ in cm. Units of dose values are eV/g per history. For each coordinates, user can select the number of bins used to report the data. Moreover, the tally reports the depth dose distribution along the z coordinate in eV/(g/cm$^2$) .

Tally variables needed in the configuration file are explained below,

- **type "SPATIAL_DOSE_DISTRIB"**: Type name of the tally.
- **xmin**: Minimum value of coordinate $x$.
- **xmax**: Maximum value of coordinate $x$. Must be greater than *xmin*.
- **nx**: Number of $x$ bins. Must be, at least, 1.
- **ymin**: Minimum value of coordinate $y$.
- **ymax**: Maximum value of coordinate $y$. Must be greater than *ymin*.
- **ny**: Number of $y$ bins. Must be at least 1.
- **zmin**: Minimum value of coordinate $z$.
- **zmax**: Maximum value of coordinate $y$. Must be greater than *ymin*.
- **nz**: Number of $z$ bins. Must be at least 1.

The variables that correspond to a coordinate values, must be doubles, while the number of bins for each coordinate must be an integrer. For plotting purposes, two values per bin coordinate are given: the low and the middle point of each bin. Output filenames end with *spatialDoseDistrib-3D.dat* for 3D absorbed dose distribution and *depth-dose.dat* for depth dose distribution. Code 72 shows an example of this tally with some lines of the configuration file *disc.in* that belongs to the example 1-disc-novr.

```
tallies/SpatialDoseDistrib/type "SPATIAL_DOSE_DISTRIB"
tallies/SpatialDoseDistrib/print-xyz true      (optional)
tallies/SpatialDoseDistrib/xmin 0.0            (mandatory)
tallies/SpatialDoseDistrib/xmax 1.0            (mandatory)
tallies/SpatialDoseDistrib/nx 1                (mandatory)
tallies/SpatialDoseDistrib/ymin 0.0            (mandatory)
tallies/SpatialDoseDistrib/ymax 1.0           (mandatory)
tallies/SpatialDoseDistrib/ny 1               (mandatory)
tallies/SpatialDoseDistrib/zmin 0.0           (mandatory)
tallies/SpatialDoseDistrib/zmax 0.005         (mandatory)
tallies/SpatialDoseDistrib/nz 100             (mandatory)
```

Code 72: Spatial Dose Distribution tally configuration

### 6.5.7 Angular Detector

This tally reports the angular energy spectrum in a specified detector. This information is tallied in the energy interval $[emin, emax)$ in eV and angular intervals $[theta1, theta2)$, $[phi1, phi2)$, in degrees. The energy spectra of the emerging particles is tallied in 1/(eV sr particle).

Below, we describe the variables used in this tally,

- **type "ANGULAR_DET "**: Tally type name.

- **detector**: Detector index where the angular detector will be calculated.
- **emin**: Minimum energy value.
- **emax**: Maximum energy value. Must be greater than *emin*.
- **theta1**: Minimum value of polar angle.
- **theta2**: Maximum value of polar angle. The polar interval must be in the range $(0, 180)$ and *theta2* must be greater than *theta1*.
- **phi1**: Minimum value of azimuthal angle.
- **phi2**: Maximum value of azimuthal angle. The azimuthal interval must be in the range $(0, 360)$ or $(-180, 180)$ and *phi2* must be greater than *phi1*.
- **nBinsE**: Number of bins energy bins.
- **linearScale**: Determines if the output measurements are expressed in linear scale (**true**) or in logarithmic scale (**false**). If scale is not specified, linear scale will be set.

Where *detector* and and the number of bins must be integers, and limits of energy and angular intervals must be doubles. Output filename for this tally ends with *spc-angdet-num.dat* where *num* is the number assigned to the detector.

The lines of code 73, correspond to an example where this tally is used, 1-disc-novr. The configuration file is named *disc.in*.

```
tallies/AngularDetector/type "ANGULAR_DET"
tallies/AngularDetector/detector 1                (mandatory)
tallies/AngularDetector/emin 0.0                  (mandatory)
tallies/AngularDetector/emax 40.5e3               (mandatory)
tallies/AngularDetector/theta1 90.0               (mandatory)
tallies/AngularDetector/theta2 180.0              (mandatory)
tallies/AngularDetector/phi1 0.0                  (mandatory)
tallies/AngularDetector/phi2 360.0                (mandatory)
tallies/AngularDetector/nBinsE 200                (mandatory)
tallies/AngularDetector/linearScale true          (optional)
```

Code 73: Angular detector tally configuration

### 6.5.8 Particle Generation

This tally reports information about the primary and secondary particles simulated. First, counts the number of primary particles that escape up bound and down bound and the number of absorbed particles. Moreover, calculates the probabilities up bound, down bound and absorbed primary and secondary particles respectively.

This tally doesn't need any information at the configuration file, only the tally type,

- **type "SECONDARY_GEN"**: Tally type name.

The output filename ends with *particleGeneration.dat*.

An example of the configuration file of this tally is shown at code 74. This line is the same for all examples where this tally will be used.

```
tallies/secondary/type "SECONDARY_GEN"
```

Code 74: Particle generation tally configuration

### 6.5.9 Spherical Dose Distribution

This tally reports the absorbed dose distribution in (eV/g) per history for each radial bin in the range of $[rmin, rmax)$, in cm. Radial total bins are denoted by *nbin*. Optionally, user can activate the boolean $print - xyz$ (true) to print different values for radial coordinate in the output file: the low value and the average value. This average value is weighted proportionally to $r^2$.

Tally information needed in the configuration file:

- **type "SPHERICAL_DOSE_DISTRIB "**: Type name of the tally.
- **print-xyz**: Can be activated, set to **true**, to print different values for radial coordinate in the output file i.e the low value and the average value. This average value is weighted proportionally to $r^2$. If the value is not specified will be set to **false**.
- **rmin**: Minimum value of radial coordinate. Must be greater than zero.
- **rmax**: Maximum value of radial coordinate. Must be greater than *rmin*.
- **nbin**: Number of radial bins. Must be at least 1.

The radial limits must be doubles and the number of bins must be an integer. The output filename ends with *sphericalDoseDistrib.dat*.

Any of the proposed examples use this tally. An example of a possible lines for this tally in a configuration file are shown at code 75.

```
tallies/SphericalDose/type "SPHERICAL_DOSE_DISTRIB)"
tallies/SphericalDose/print-xyz true              (optional)
tallies/SphericalDose/rmin 0.0                    (mandatory)
tallies/SphericalDose/rmax 0.06                   (mandatory)
tallies/SphericalDose/nbin 65                     (mandatory)
```

Code 75: Spherical dose distribution tally configuration

### 6.5.10 Phase Space File (PSF)

This tally creates a particle phase space file which store all particles that impact at the specified detector. Also, the stored particles can be limited by a energy range.

Tally information needed in the configuration file:

- **type "PSF "**: Type name of the tally.
- **detector**: Detector index where this tally will be calculated.
- **emin**: Minimum energy value.
- **emax**: Maximum energy value. Must be greater than *emin*.

```
tallies/psf/type "PSF"       (mandatory)
tallies/psf/detector 1       (mandatory)
tallies/psf/emin 0.0         (mandatory)
tallies/psf/emax 6.1e6       (mandatory)
```

Code 76: Phase space file tally configuration

This tally can be used with multiple threads. In that case, each thread will store their psf particles in an independent file. Then, when the simulation finishes, all files will be concatenated in thread ID ascending order.

## 6.6 Variance Reduction

PenRed main implements the very same variance reduction techniques as original FORTRAN code. These are, for generic simulations, interaction forcing (*IF*), x-ray and bremsstrahlung splitting. Then, phase space file based simulations adds particle splitting and Russian Roulette techniques. Note that variance reduction for phase space file will be configured via the corresponding particle source parameters, as we saw on section 6.2. The other techniques are configured using the prefix *VR*, as code 77 shows. There, *vr-technique* can be interaction forcing (*IForcing*), bremsstrahlung splitting (*bremss*) or x-ray splitting (*x-ray*). Then, on *objectToApply*, the user can select *bodies* or *materials* to apply the variance reduction technique to a single body or an entire material respectively. Next, *name* is a text identifier for this VR technique. Finally, *parameter/path* and *value* depends on the VR type.

```
VR/vr-technique/objectToApply/name/parameter/path/path value
```

Code 77: Variance reduction configuration pattern

First, lets see the interaction forcing (IF) configuration. Code 78 shows a complete interaction forcing configuration used on example *4-x-ray*. As we can see, this example sets interaction forcing to a single body, which alias is specified by the parameter *body*. Then sets the kind of particle to force using the parameter *particle* and the interaction numerical identifier to force (*interaction*). That identifier can be found on section 5.4. Next paramter, *factor* sets the interaction forcing amplification factor and, finally, *min-weight* and *max-weight* limits the weight window where apply this VR technique.

```
VR/IForcing/bodies/VR1/body "1"
VR/IForcing/bodies/VR1/particle "electron"
VR/IForcing/bodies/VR1/interaction 2
VR/IForcing/bodies/VR1/factor 400
VR/IForcing/bodies/VR1/min-weight 0.1
VR/IForcing/bodies/VR1/max-weight 2
```

Code 78: Interaction forcing configuration for bodies

To set the same interaction forcing on one materials, simply substitute *bodies* by *materials*, *body* parameter by *mat-index* and set the material index where apply VR as integer value. For example, to apply this IF on material 2 the configuration should look like code 79.

```
VR/IForcing/materials/VR1/mat-index 2
VR/IForcing/materials/VR1/particle "electron"
VR/IForcing/materials/VR1/interaction 2
VR/IForcing/materials/VR1/factor 400
VR/IForcing/materials/VR1/min-weight 0.1
VR/IForcing/materials/VR1/max-weight 2
```

Code 79: Interaction forcing configuration for materials

Next technique, bremsstrahlung splitting, requires one or two parameters depending on whether the VR is specified for bodies or materials respectively. Both patterns are shown at code 80, where *body-alias* must be substitute by the body alias where splitting want to be applied, *splitting-factor* specify the number of times bremsstrahlung photons will be cloned and *imat* is the material index where apply splitting.

```
VR/bremss/bodies/body-alias/splitting splitting-factor

VR/bremss/materials/mat-index imat
VR/bremss/materials/splitting splitting-factor
```

Code 80: Bremsstrahlung splitting configuration for bodies and materials

Finally, x-ray splitting uses a pattern analogous to bremsstrahlung case, exemplified at code 81.

```
VR/bremss/bodies/body-alias/splitting splitting-factor

VR/bremss/materials/mat-index imat
VR/bremss/materials/splitting splitting-factor
```

Code 81: Bremsstrahlung splitting configuration for bodies and materials

## 6.7 Simulation parameters

Simulation parameters are main specific, unlike source, tally, geometry or material configurations. This section will explain the available parameters for “pen_main” program. All parameters follows the pattern shown at code 82.

```
simulation/parameter/path value
```

Code 82: Simulation parameters configuration pattern

Our main provide the capability to dump the current state of the whole simulation. This dump can be used to resume a crashed simulation. To configure this feature, the user can use a set of configuration parameters. The first one, *dump-interval* specify the time, in seconds, between successive dumps. Next, *dump2read* expects a string with the name of a dump file to read. This file will be read before the simulation beginning to continue a previous simulation. Another parameter is *dump2write* which expect a string and allow the user to change the default dump filename (dump.dat). Finally, *dump2ascii* tells to the program that the simulation should not be resumed. Instead, the program will load the state stored in the specified dump file (specified with *dump2read*) and extract the tally contents using the usual data reports. Notice that to use this option is necessary to specify the *dump2read* parameter. An example of dump configuration is shown at code 83, where the simulation will resume the stored state at dump file *dump.dat* and store the new generated dumps to *dump2.dat*. New dumps will be generated every 3600 $s$.

```
simulation/dump-interval 3600
simulation/dump2read "dump.dat"
simulation/dump2write "dump2.dat"
```

Code 83: Dump configuration parameters

When multiple threads are used during the simulation, the program will create a independent dump file for each one. To avoid override the dump files, each thread appends an identifier using its thread ID number, same as used for tally reports. Also, when we use multi-threading to read a dump file, the program expects one dump file for each thread with the name specified by the *dump2read* parameter and the thread ID prefix.

Another type of simulation parameters is the one that allow us to control the multi-threading capabilities. The first parameter to configure multi-threading is *nthreads*, which expect an integer value to specify the number of threads to use within the simulation. The number of histories to simulate on each source will be distributed with all specified threads. By default the number of threads is set to one.

```
simulation/threads 2
```

Code 84: Number of threads specification

Notice that MPI can be combined with multi-threading. The number of MPI processes is not specified at the configuration file, instead is specified via the *mpirun* parameters (see section 2). So, when both kinds of parallelism are combined, each MPI process will spawn a number of threads equal to the specified at the configuration file. This approach "suggests" to the user to create a single MPI process for each node in a distributed memory infrastructure and use threads instead of more MPI processes. With this method, each node will use less memory because data bases will be shared by all threads in the same node. Furthermore, the memory access will be more efficient due to the memory sharing between threads.

Actually is not possible to specify a different number of threads for each MPI process. This feature is intended to be implemented at future PenRed versions.

Following with multi-threading parameters, *thread-affinity* parameter expect a bolean to enable or disable CPU affinity. This feature can be used only if threads are implemented via *pthreads* package.

```
simulation/thread-affinity true
```

Code 85: Threads affinity

Finally, the user can specify the initial random generator seeds using *seed*1 and *seed*2 parameters. Another option is select a seed pair provided by *rand0* [8] function via the *seedPair* parameter. Notice that on multi-threading and/or MPI simulations, only the parameter *seedPair* can be used to set initial seeds. This restriction is necessary to ensure that each random number chain is truly independent.

```
simulation/seed1 1
simulation/seed2 1
simulation/seedPair 12
```

Code 86: Initial seeds

## 6.8 Examples

**Under construction...**

PenRed has been tested with the original code PENELOPE on which it is based via the provided PENELOPE examples. Some parts of the configuration files of these examples have been shown at previous sections to exemplify the different samplers options, source, materials, geometry and tally definitions etc. At this section, all of the examples included at the distributed package are briefly described.

To execute the examples, geometry file, configuration file and material files are required. They must be at the same directory. Moreover, the executable file created after the program compilation is needed. As we have explained at section 2, the executable is compiled at *src/compiled/mains/pen_main*. To run the simulation, the executable must be copied at the same folder where the example will be run with the other required files. After that, user can start the execution as follows,

*./pen_main path/to/configuration/file*

### 6.8.1 1-disc

This example presents a point souce of electrons and a homogeneous disc phantom of $Cu$. The source is a monoenergetic beam with energy of $40KeV$ and it is located at $(x, y, z) = (0, 0, -0.0001)$ cm. The phantom size is a radius of 0.01 cm at plane $XY$ and height of 0.005 cm at $z$ axis with the base at $z = 0$ cm.

#### 6.8.1.1 1-disc-no-vr

This is the first version of the disc example, without variance reduction techniques. The required material files to execute this example is only *Cu.mat* file, and the configuration file is named disc.in. The geometry is read from the geometry file named *disc.geo*. The output of this example execution are the cylindrical and spatial dose distribution, the emerging particles distribution and the impact detector tallies for fluence and energy spectrum.

#### 6.8.1.2 1-disc-vr

The second version of the disc example includes variance reduction techniques. The material and geometry file are the same than the first version and, although the configuration file has the same name, there are some differences between these two versions. This second one is an example of the interaction forcing, x-ray and bremsstrahlung splitting capabilities of the PenRed code. At the configuration file is specified which is the body that suffers the variance reduction techniques and the kind of particle and interaction to force, electrons in this case. The output of this example execution is the same than at the first version. Notice that, when variance reduction techniques are used, the number of histories of the simulation is reduced because of the splitting applied to them.

## References

[1] Salvat F., Penelope. a code system for monte carlo simulation of electron and photon transport, Issy-Les-Moulineaux: OECD Nuclear Energy Agengy (2014).

[2] F. Salvat, PENELOPE-2018: A code System for Monte Carlo Simulation of Electron and Photon Transport, OECD/NEA Data Bank, Issy-les-Moulineaux, France, 2019, available from `http://www.nea.fr/lists/penelope.html`.

[3] Dcmtk, `https://github.com/DCMTK/dcmtk`, accessed: 2019-09-28.

[4] R. L. Graham, T. S. Woodall, J. M. Squyres, Open mpi: A flexible high performance mpi, in: R. Wyrzykowski, J. Dongarra, N. Meyer, J. Waśniewski (Eds.), Parallel Processing and Applied Mathematics, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 228–239.

[5] W. Gropp, Mpich2: A new start for mpi implementations, in: Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer-Verlag, Berlin, Heidelberg, 2002, p. 7.

[6] F. James, A review of pseudorandom number generators, Comput. Phys. Commun. 60 (1990) 329–344.

[7] V. Giacometti, S. Guatelli, M. Bazalova-Carter, A. Rosenfeld, R. Schulte, Development of a high resolution voxelised head phantom for medical physics applications, Physica Medica 33 (2017) 182 – 188. `doi:https://doi.org/10.1016/j.ejmp.2017.01.007`.
URL `http://www.sciencedirect.com/science/article/pii/S1120179717300078`

[8] A. Badal, J. Sempau, A package of linux scripts for the parallelization of monte carlo simulations, Computer Physics Communications 175 (6) (2006) 440 – 450. `doi:https://doi.org/10.1016/j.cpc.2006.05.009`.
URL `http://www.sciencedirect.com/science/article/pii/S001046550600230X`