

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

V. Giménez-Alventosa¹, V. Giménez Gómez², and S. Oliver³

¹Instituto 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

²Departament de Física Teòrica and IFIC, Universitat de València-CSIC, Dr. Moliner, 50, 46100, Burjassot, València, Spain, vicente.gimenez@uv.es

³Instituto de Seguridad Industrial, Radiofísica y Medioambiental (ISIRYM), Universitat Politècnica de València, Camí de Vera s/n, 46022, València, Spain, sanolgi@upvnet.upv.es

July 16, 2021


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 concepts of PenRed structure and contains all the necessary information to use the framework and extend it creating custom modules or extensions of the available 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

V. Giménez-Alventosa  <https://orcid.org/0000-0003-1646-6094>

V. Giménez Gómez  <https://orcid.org/0000-0003-3855-2567>

S. Oliver  <https://orcid.org/0000-0001-8258-3972>

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 ones. 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 PENGEO package of PENELOPE, PENCT program and PENEASY. In addition, contains a DICOM geometry class to handle DICOM images as geometries to be simulated. 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 are described in section 4.1. Some both auxiliary 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 (section 8) can be found in the folder **utilities**, such as 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) using all the available PenRed components, such as geometry, tallies and source types. In addition, several variance reduction techniques can be employed as interaction forcing and particle splitting (section 6.5).

The manual is organized as follows. The 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 structure format used to configure the simulations. Then, section 4 is only intended for user which want to extend the PenRed functionality, thus is not required to use the framework as is. This one 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, is required some knowledge about classes, inheritance, template functions as well as other C++ features of the 2011 standard. In section 5 a briefly description of important definitions, constants, parameters and units is shown. Following, section 6, shows how to execute the main program provided within the PenRed package, and describes all the implemented tallies, sources, geometries, variance reduction techniques and other parameters to configure the simulation. Then, in the section 7, the examples located in the **examples** folder, which are ready to be simulated, are described. Finally, the offered utilities in the PenRed package and how to use them are explained in the section 8.

For users only interested in package usage, go to sections 2, 3 and 6, which explain the installation procedure, the internal data format used in the 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

1	Introduction	1
2	Installation and execution	6
2.1	Linux instalation	6
2.2	Windows instalation	7
2.3	Basic usage	11
2.4	Optional features	11
3	Internal data format	12
3.1	Implementation	12
4	Engine components	13
4.1	Kernel Modules	13
4.1.1	Random generator	13
4.1.2	Particle states	14
4.1.3	State Stacks	14
4.1.4	Grids	15
4.1.5	Materials	15
4.1.6	Contexts	16
4.1.7	Particles	19
4.1.8	Interactions	24
4.2	Geometries	25
4.2.1	Objects based geometries	28
4.2.2	Mesh geometries	29
4.3	String instantiator mechanism	32
4.4	State samplers	34
4.4.1	Spatial Samplers	34
4.4.2	Direction Samplers	38
4.4.3	Energy Samplers	38
4.4.4	Time Samplers	38
4.4.5	Specific Samplers	39
4.4.6	Generic state generators	40
4.4.7	Specific state generators	43
4.5	Tallies	44
4.6	Variance reduction	53
4.7	Load Balance	57
5	Constants, parameters and definitions	58
5.1	Important definitions	58
5.2	Units	60
5.3	Particle indexes	60
5.4	Particle interaction indexes	60
5.5	Atomic electron shells	61
6	Framework usage	61
6.1	Materials	61
6.1.1	Configuration	62
6.2	Particle sources	63

6.2.1	Spatial source samplers	65
6.2.2	Direction samplers	66
6.2.3	Energy samplers	67
6.2.4	Time samplers	68
6.2.5	Specific Sources	68
6.3	Geometries	69
6.3.1	Quadric	69
6.3.2	Voxel	70
6.3.3	DICOM	72
6.4	Tallies	74
6.4.1	Radial and Cylindrical Dose Distribution	74
6.4.2	Emerging Particles Distribution	75
6.4.3	Energy Deposition Body	76
6.4.4	Energy Deposition Material	76
6.4.5	Impact Detector	77
6.4.6	Spatial Dose Distribution	79
6.4.7	Angular Detector	79
6.4.8	Particle Generation	80
6.4.9	Spherical Dose Distribution	81
6.4.10	Phase Space File (PSF)	81
6.4.11	Kerma track length estimator	82
6.5	Variance Reduction	84
6.5.1	Context specific	84
6.5.2	Specific VR techniques	85
6.5.3	Generic VR techniques	86
6.6	Simulation parameters	87
6.7	Load balance parameters	88
7	Examples	89
7.1	1-disc	89
7.1.1	1-disc-no-vr	90
7.1.2	1-disc-vr	90
7.2	2-plane	90
7.3	3-detector	90
7.4	4-x-ray-tube	90
7.5	5-accelerator	91
7.5.1	5-accelerator-1	91
7.5.2	5-accelerator-2	91
7.5.3	5-accelerator-3	91
7.6	6-polarisation	92
7.7	7-aba	92
7.8	8-fake-chamber	92
7.8.1	8-fake-chamber-novr	92
7.8.2	8-fake-chamber-vr	92
8	Utilities	92
8.1	Mutren	92
8.2	iaeaPSF	93
8.3	geo2voxel	93
8.4	range	94
8.5	PSF spectrum	95

8.6	PSF to ASCII	95
8.7	Registers	95
A	Predefined materials	96

2 Installation and execution

Some PenRed features 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++ with version greater than GCC 4.7.3 with support for C++ 11 standard. However, PenRed can also be compiled with Intel icc version 2019, clang version 8 and MSVS 2019.
- CMake minimal version 3.4.

2.1 Linux instalation

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, which includes a CMake file and a bash script (`compile.sh`) to simplify the installation and compile the code automatically. In this script, you can enable/disable the following main 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].
- Load balance: Enables load balancing system between threads and MPI processes. This option requires, as least, multi-threading capabilities, because uses threads to handle MPI communications.

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
```

Once the dependencies are installed, the compilation can be done 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" -DWITH_LB="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

2.2 Windows instalation

To compile the PenRed code using MSVS on windows, first, select “Clonea repository” from the Visual Studio start window (Figure 1).

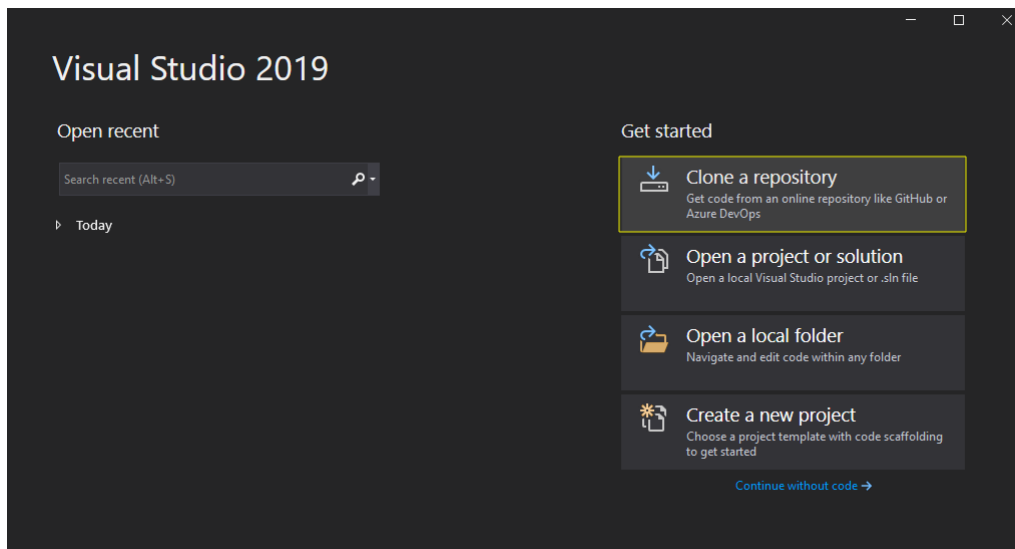


Figure 1: Clone a repository in MSVS

Then, set the PenRed repository url and push on the clone button (Figure 2).

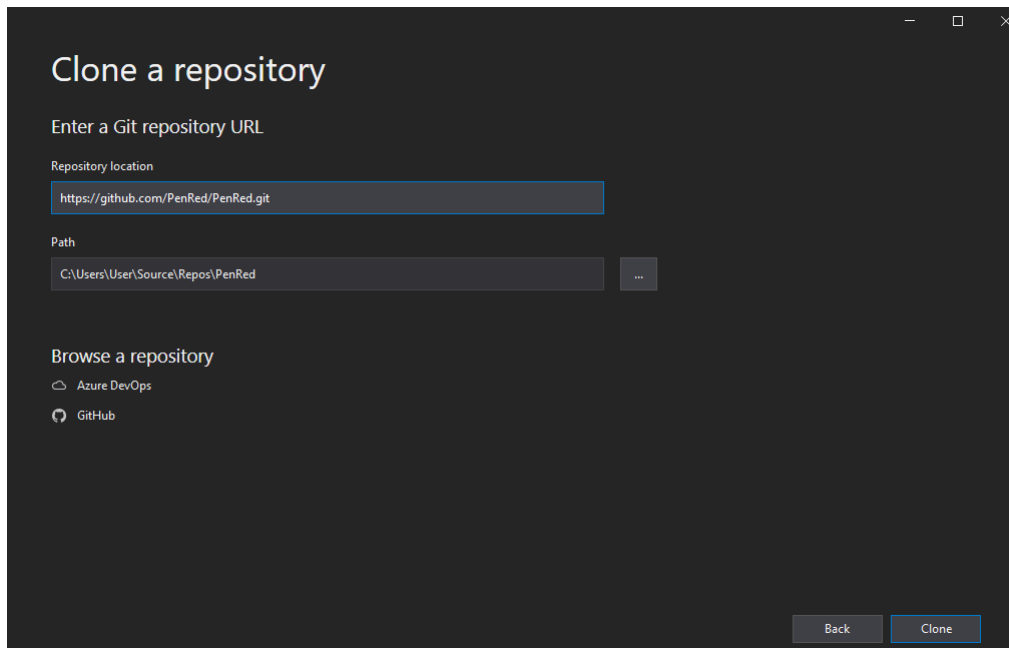


Figure 2: Clone PenRed repository with MSVS

The download will start automatically. Once the Cmake configuration ends, to avoid compiling with a debug profile, add a new configuration with the configuration manager (Figure 3).

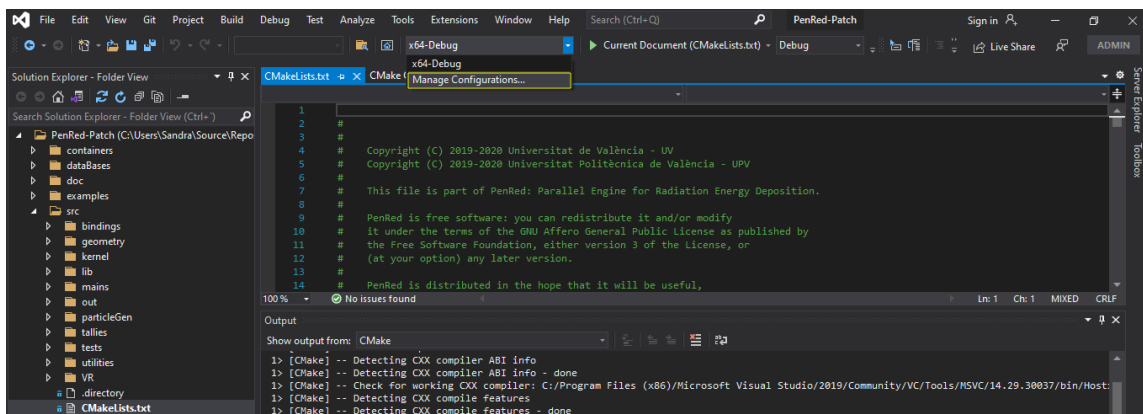


Figure 3: Manage MSVS configurations

In the configuration panel to the left, click on the button with the green *plus* sign to add a new configuration and select the release depending on your system. In the following image, we selected a release for 64-bit system (Figure 4).

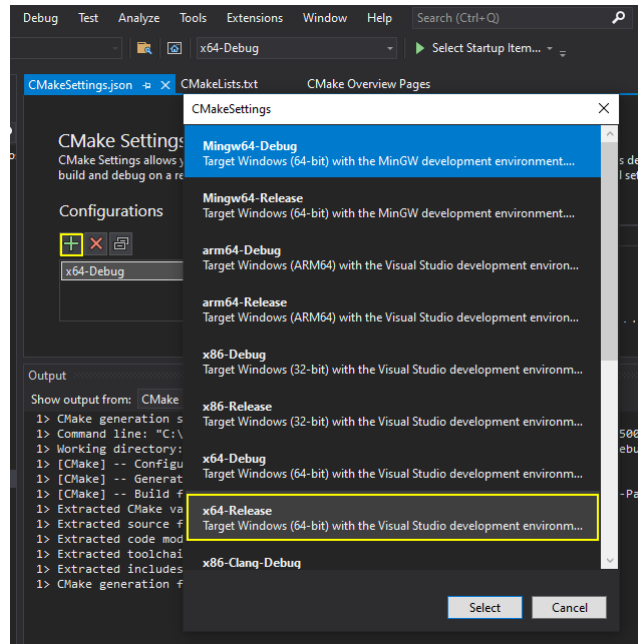


Figure 4: Add configuration in MSVS

Once you have selected the new configuration, and push on “Save and generate CMake cache to load variables” to be able to change the CMake variables for this configuration and compile it (Figure 5).

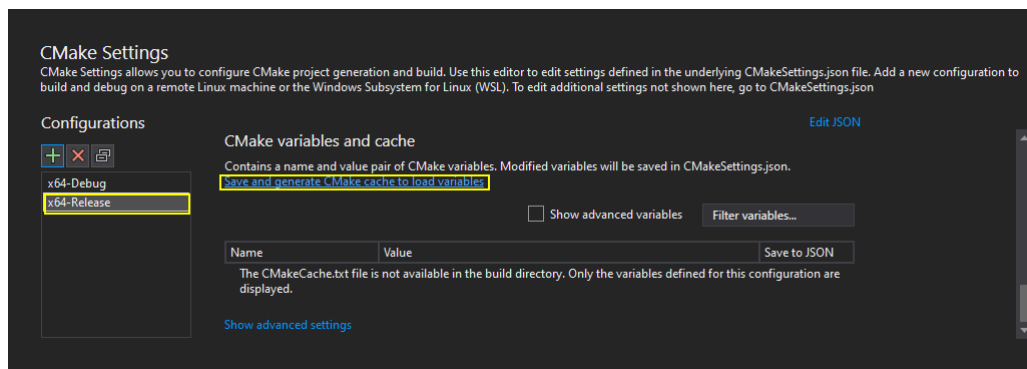


Figure 5: Select configuration in MSVS

Now, we can change all the Cmake configuration variables to enable or disable MPI support and other features (Figure 6).

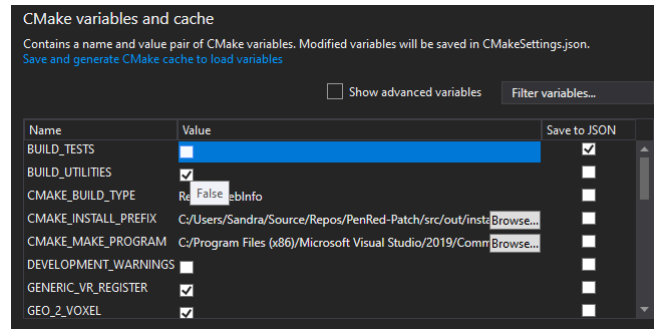


Figure 6: Set compilation flags in MSVS

Finally, build and install PenRed (Figure 7).

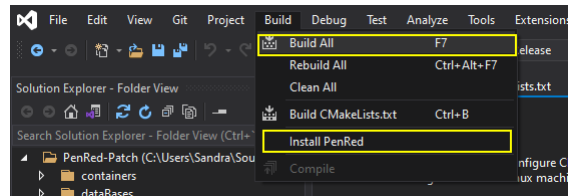


Figure 7: Build and install PenRed in MSVS

If the compilation finishes successfully, a new folder named *compiled* will be created inside the *src* folder containing the PenRed's main program and the executable for all other enabled utilities (Figure 8).

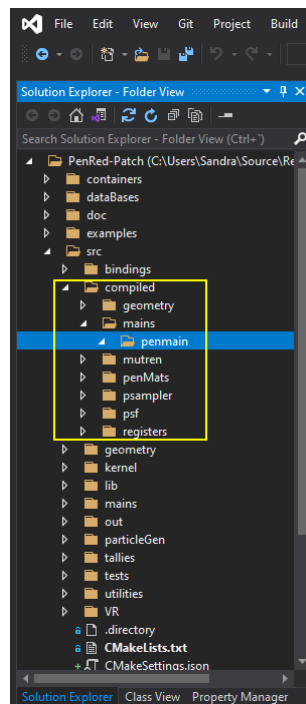


Figure 8: PenRed compiled programs

2.3 Basic usage

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. All the details regarding the simulations configuration can be found in the section 6. Moreover, the *examples* folder contain several configuration file examples with the corresponding material and geometry files ready to be executed (section 7). 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 as specify the hosts where execute the code via the *hostfile* option.

2.4 Optional features

The following list summarise the optional compilation flags available during the PenRed compilation procedure.

- **BUILD_UTILITIES:** Enables or disable the compilation of utilities, such as *geo2vox*, sampler, geometry and tally registers or load balance server. If enabled, the compilation of each utility can be enabled or disabled individually.
- **WITH_MULTI_THREADING:** Handles multithreading capabilities.
- **WITH_MPI:** Handles MPI capabilities. Require an MPI library installed.
- **WITH_NATIVE:** Switch on or off the compilation with native optimizations. Actually, this flag only works on GCC and Intel C++ compilers.
- **WITH_DICOM:** Handles DICOM capabilities. Requires the *dcmtk* library.
- **WITH_LB:** Enables or disables load balance capabilities. If enabled, some optional features related with the balance system can be configured:
 - **WITH_SSL:** Enables or disables SSL secure connections for TCP or HTTPS communications. Requires OpenSSL libraries and its dependences.
 - **WITH_HTTP:** Enables or disables connections to a HTTP/HTTPS rest api as balance server (Experimental feature). Requires the libCurl library.
- **BUILD_TESTS:** Enables or disable the compilation of test codes.
- **DEVELOPMENT_WARNINGS:** Enables or disables development warning flags.

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 in 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 quotes ("text") to be parsed correctly. For example, code 1 shows a configuration for a cylindrical dose distribution tally.

```
1 tallies/cylDoseDistrib/type "CYLINDRICAL_DOSE_DISTRIB"
2 tallies/cylDoseDistrib/print-xyz true
3 tallies/cylDoseDistrib/rmin 0.0
4 tallies/cylDoseDistrib/rmax 30.0
5 tallies/cylDoseDistrib/nbinsr 60
6 tallies/cylDoseDistrib/zmin 0
7 tallies/cylDoseDistrib/zmax 30.0
8 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 of 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 used to 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.

```

1 //Base material definition. Material_A is derived
2 //directly from abstract material class "abc_material"
3 class Material_A : public abc_material{ ... };
4 //Derived material definition
5 class Material_B : public Material_A{ ... };
6
7 //Context definition selecting "Material_A" as base material type
8 class Context_A : public abc_context<Material_A>{ ... };
9
10 int main() {
11 ...
12
13 //Set number of materials
14 size_t nMats = 5;
15 //Create the context
16 Context_A context;
17 //Set material type "Material_B" for our context
18 context.setMats<Material_B>(nMats);
19
20 //Instead, to use base class material we should use
21 //context.setMats<Material_A>(nMats);
22 ...
23 }

```

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 cutoff 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*).

```

1
2 //Define a particle class "particle_A" which uses as particle
3 //state type the "state_type" class , as a compatible context the
4 //"context_type" class and, finally , the "material_type" class as
5 //compatible material.
6 class particle_A : public abc_particle<state_type , context_type , material_type>{
7
8 ...
9
10 public :
11
12 //Define the minimum class constructor
13 particle_A(const context_type& contextIn) : abc_particle(contextIn ,
14     PARTICLE_INDEX,NINTERACTIONS,0.0) {...}
15 ...
16
17 };

```

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 three 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 losses. 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,

- **DSeff**: Returns *dseff* 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 *dseff*, *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 them 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 find the three PENELOPE particle implementations: electron, gamma and positron.

4.1.7.1 Adding particles

To add particles in PenRed, the user must include 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 code 4.

```

1 //Particle index enumeration
2
3 enum pen_KPAR{
4     PEN_ELECTRON,
5     PEN_PHOTON,
6     PEN_POSITRON,
7
8     NEW_PARTICLE, // <— Add a identifier of our new particle
9
10    ALWAYS_AT_END // <— Never add an identifier after that
11 };
12
13 //Particles index to string function
14 inline const char* particleName(const unsigned kpar){
15     switch(kpar){
16         case PEN_ELECTRON: return "electron";
17         case PEN_PHOTON: return "gamma";
18         case PEN_POSITRON: return "positron";
19
20         //Add a text name for our particle
21         case NEW_PARTICLE: return "newName";
22
23         default: return nullptr;
24     }
25 }

```

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.

```

1      void myGeometry::step(pen_particleState& state ,
2                             double DS, double &DSEF,
3                             double &DSTOT, int &NCROSS) const{
4
5          //Check if the particle comes from outside of geometry
6          if(state.IBODY >= getBodies()){
7              //Particle come from outside the system,
8              //DS value is ignored
9              moveToGeometry( state ,DSEF,NCROSS);
10             DSTOT = DSEF;
11             return;
12         }
13
14         //The particle is in the geometry system
15
16         //Get the distance until next interface and the
17         //corresponding body and material indexes
18         unsigned newIBODY, newMAT;
19         double dsToInterface = nextInterface( state ,
20                                                newIBODY,
21                                                newMAT);
22
23         if(dsToInterface > DS){
24             //The particle will be moved freely
25             state.X += state.U*DS;
26             state.Y += state.V*DS;
27             state.Z += state.W*DS;
28         }

```

```

29         DSEF = DS;
30         DSTOT = DSEF;
31         NCROSS = 0; //No interface crossed
32         return;
33     }
34     else{
35         //The particle find an interface
36         DSEF = dsToInterface;
37         NCROSS = 1; //An interface crossed
38         //Check if the material behind is void
39         if(newMAT == 0){
40             //Is a void region, find next interface or
41             //escape from geometry
42             unsigned newIBODY2, newMAT2;
43             double dsToNonVoid = nextInterface(state,
44                                                 newIBODY2,
45                                                 newMAT2);
46
47             if(newIBODY2 < getBodies()){
48                 //The particle has crossed another
49                 //interface and, for instance,
50                 //remains in the geometry system
51                 NCROSS += 1;
52             }
53
54             DSTOT = DSEF + dsToNonVoid;
55
56             state.X += state.U*DSTOT;
57             state.Y += state.V*DSTOT;
58             state.Z += state.W*DSTOT;
59             state.IBODY = newIBODY2;
60             state.MAT = newMAT2;
61         }
62         else{
63             //Is a non void region
64             DSTOT = DSEF;
65             state.X += state.U*DSEF;
66             state.Y += state.V*DSEF;
67             state.Z += state.W*DSEF;
68             state.IBODY = newIBODY;
69             state.MAT = newMAT;
70         }
71     }
72 }
73 }
74

```

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 cutoff 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 cutoff 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.

```
1 #ifndef _PEN_DUMMY_GEOMETRY_
2 #define _PEN_DUMMY_GEOMETRY_
3
4 #include "geometry_classes.hh"
5
6 class pen_dummyGeo : public abc_geometry<pen_baseBody>{
```

```

7 DECLARE_GEOMETRY(pen_dummyGeo)
8
9 public:
10 pen_dummyGeo() {
11     configStatus = 0;
12 }
13
14 int configure(const pen_parserSection& /*config*/,
15              unsigned /*verbose*/){
16     return 0;
17 }
18 void locate(pen_particleState& const;
19 void step(pen_particleState&,
20          double,
21          double &,
22          double &,
23          int &) const;
24
25 inline unsigned getIBody(const char* /*name*/) const {
26     return getElements();
27 }
28 };
29
30 #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.

```

1 #define DECLARE_GEOMETRY( Class ) \
2 public: \
3 inline int registerStatus() const { return ___register_return; } \
4 inline const char* readID() const { return ___ID; } \
5 private: \
6 static const char* ___ID; \
7 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.

```

1 #include "dummy-geo.hh"
2
3 void pen_dummyGeo::locate(pen_particleState& /*state*/) const {
4 }
5
6 void pen_dummyGeo::step(pen_particleState& /*state*/, double /*DS*/, double &
  DSEF, double &DSTOT, int &NCROSS) const {
7
8     DSEF = 0.0;
9     DSTOT = 0.0;
10    NCROSS = 0;
11 }
12
13 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.

```

1 #define REGISTER_GEOMETRY(Class, ID) \
2     const int Class::__register_return = penGeoRegister.add<Class>(static_cast<\
  const char *>(#ID)); \
3     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 code 13. This mechanism is described in detail at the string instantiation 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 codes 10 and 11 respectively.

```

1 #ifndef __PENELOPE_OBJECT_GEOMETRIES__
2 #define __PENELOPE_OBJECT_GEOMETRIES__
3
4 //Include implemented geometries include files
5 #include "quadric-geo.hh"
6 #include "dummy-geo.hh" // <— Our new geometry
7
8 #endif

```

Code 10: Geometry include file

```

1 //Include implemented geometries source files
2 #include "quadric-geo.cpp"
3 #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.

```

1 template <class mother, class sub>
2 void instanceInheritance(mother*& pmoth){
3     sub* psub = new sub;
4     pmoth = nullptr;
5     pmoth = dynamic_cast<mother*>(psub);
6     if(pmoth == nullptr){
7         delete psub;
8     }
9 }

```

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,

$$std :: function < void(motherClass * \&) > \quad (1)$$

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

$$\begin{aligned} & template < class motherClass, class subclass > \\ & void instanceInheritance(motherClass * \&pmoth) \end{aligned} \quad (2)$$

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

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

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

```

1  motherClass* createInstance(std::string typeId) {
2
3  //Lock the mutex
4  std::lock_guard<std::mutex> guard(lock);
5
6  //Search specified ID
7  if(creatorsMap.find(typeID) != creatorsMap.end()){
8      motherClass* pmoth = 0;
9      creatorsMap[typeID](pmoth);
10     return pmoth;
11 }
12
13 return nullptr;
14 }
```

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,

- **registered:** 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,

$$X' = T + R \cdot X, \quad (4)$$

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 *setRotationZZY*, which takes as arguments three angles and create the rotation matrix as the product of three rotations around the axis Z , Y , Z ,

$$R(\alpha_1, \alpha_2, \alpha_3) = R_z(\alpha_1)R_y(\alpha_2)R_z(\alpha_3) \quad (5)$$

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.

```

1
2 #ifndef __BOX_SPATIAL_SAMPLING__
3 #define __BOX_SPATIAL_SAMPLING__
4 class box_spatialSampling : public abc_spatialSampler {
5     DECLARE_SAMPLER(box_spatialSampling)
6
7 private:
8
9     double dx, dy, dz;
10
11 public:
12
13     box_spatialSampling() : dx(1.0),
14                             dy(1.0),
15                             dz(1.0)
16     {}
17
18     void geoSampling(double pos[3], pen_rand& random) const;
19
20     int configure(const pen_parserSection& config,
21                  const unsigned verbose = 0);
22
23 };
24
25 #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,

$$x = dx \cdot \xi - dx/2 = dx (\xi - 0.5) \quad (6)$$

where ξ 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.

```

1 #include "box_spatialSampling.hh"
2
3 void box_spatialSampling::geoSampling(double pos[3], pen_rand& random) const{
4
5     pos[0] = dx*(random.rand()-0.5);
6     pos[1] = dy*(random.rand()-0.5);
7     pos[2] = dz*(random.rand()-0.5);
8
9 }
10
11 int box_spatialSampling::configure(const pen_parserSection& config, const
    unsigned verbose){
12
13     int err;
14
15     err = config.read("size/dx", dx);
16     if(err != INTDATA.SUCCESS){
17         if(verbose > 0){
18             printf("boxSpatial:configure:unable to read 'size/dx' in configuration.
                Double expected\n");
19         }
20         return -1;
21     }
22
23     err = config.read("size/dy", dy);
24     if(err != INTDATA.SUCCESS){
25         if(verbose > 0){
26             printf("boxSpatial:configure:unable to read 'size/dy' in configuration.
                Double expected\n");
27         }
28         return -1;
29     }
30
31     err = config.read("size/dz", dz);
32     if(err != INTDATA.SUCCESS){

```

```

33     if(verbose > 0){
34         printf("boxSpatial:configure:unable to read 'size/dz' in configuration.
35         Double expected\n");
36     }
37     return -1;
38 }
39 if(dx < 0.0 || dy < 0.0 || dz < 0.0){
40     return -2;
41 }
42
43 err = config.read("position/x",translation[0]);
44 if(err != INTDATA.SUCCESS){
45     if(verbose > 0){
46         printf("boxSpatial:configure:unable to read 'position/x' in
47         configuration. Double expected\n");
48     }
49     return -2;
50 }
51 err = config.read("position/y",translation[1]);
52 if(err != INTDATA.SUCCESS){
53     if(verbose > 0){
54         printf("boxSpatial:configure:unable to read 'position/y' in
55         configuration. Double expected\n");
56     }
57     return -2;
58 }
59 err = config.read("position/z",translation[2]);
60 if(err != INTDATA.SUCCESS){
61     if(verbose > 0){
62         printf("boxSpatial:configure:unable to read 'position/z' in
63         configuration. Double expected\n");
64     }
65     return -2;
66 }
67 if(verbose > 1){
68     printf("Box center (x,y,z):\n %12.4E %12.4E %12.4E\n",translation[0],
69     translation[1],translation[2]);
70     printf("Box size (dx,dy,dz):\n %12.4E %12.4E %12.4E\n",dx,dy,dz);
71 }
72 return 0;
73 }
74
75 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.

```

1 #ifndef __PENRED.SPATIAL.SAMPLERS__
2 #define __PENRED.SPATIAL.SAMPLERS__
3
4 #include "box_spatialSampling.hh" // <— Our new sampler
5 #include "point_spatialSampling.hh"
6

```

```
7 #endif
```

Code 16: Spatial samplers include files

```
1 #include "box_spatialSampling.cpp" // <— Our new sampler
2 #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 *E_{max}*. 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 code 18. The first one is *E_{max}*, 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.

```

1  int configure(double& Emax,
2      const abc_spatialSampler* pSpatial,
3      const abc_directionSampler* pDirection,
4      const abc_energySampler* pEnergy,
5      const abc_timeSampler* pTime,
6      const pen_parserSection& config,
7      const unsigned verbose);

```

Code 18: Configure method for specific samplers

Respect to the sampling method (*sample*) 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-safe 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.

```

1  void sample(pen_particleState& state,
2      pen_KPAR& genKpar,
3      unsigned long& dhist,
4      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.

```

1 enum __usedSamp{
2     USE_SPATIAL      = 1 << 0,
3     USE_DIRECTION    = 1 << 1,
4     USE_ENERGY       = 1 << 2,
5     USE_TIME         = 1 << 3,
6     USE_GENERIC      = 1 << 4,
7     USE_NONE         = 1 << 5
8 };

```

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 \mid operator. For example, to delegate the spatial and directional sampling to generic samplers we will use something like lines code 21.

```

1 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.

```

1 instantiator<abc_spatialSampler>& pen_genericStateGen::spatialSamplers() {
2     static instantiator<abc_spatialSampler>* ans =
3         new instantiator<abc_spatialSampler>();
4     return *ans;
5 }

```

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 code 23.

```

1     template <class subclass>
2     static int addSpatialSampler(const char* typeID){
3         return spatialSamplers().addSubType<subclass>(typeID);
4     }
5

```

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.

```

1 void pen_genericStateGen::sample(pen_particleState& state ,
2                                 pen_rand& random) const{
3
4     //Reset state
5     state.reset();
6
7     //Perform time sampling if exists
8     if(timeSampler != nullptr){
9         timeSampler->sample(state , random);
10    } else {
11        state.PAGE = 0.0;
12    }
13
14    state.LAGE=LAGE;
15
16    //Perform direction sampling
17    directionSampler->sample(state , random);
18
19    //Perform spatial sampling and locate the particle
20    spatialSampler->sample(state , random);
21    geometry->locate(state);
22
23    //Check if source is restricted to specified body
24    //or material
25    if(sourceBody >= 0){
26        while(state.IBODY != (unsigned)sourceBody){
27            spatialSampler->sample(state , random);
28            geometry->locate(state);
29        }
30    }
31    else if(sourceMat > 0){
32        while(state.MAT != sourceMat){
33            spatialSampler->sample(state , random);
34            geometry->locate(state);
35        }
36    }
37
38    //Perform energy sampling
39    energySampler->sample(state , random);
40
41    //Its a primary (source) particle , set ILB[0] = 1
42    state.ILB[0] = 1;
43 }

```

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 the code 25.

```
1  void sample(particleState& state ,
2             pen_KPAR& genKpar ,
3             unsigned long& dhist ,
4             const unsigned thread ,
5             pen_rand& random){
6
7     genKpar = kpar;
8
9     //Check if generic sampling is required
10    if(useGeneric){
11        //Perform generic sampling
12        dhist = 1;
13        genericGen.sample(state ,random);
14
15        //If specified , perform specific sampling
16        if(useSpecific){
17            specificSamplerVect[thread]->
18                sample(state ,genKpar ,dhist ,random);
19        }
20    }
21    else{ //No generic sampling
22        //If specified , perform specific sampling
23        if(useSpecific){
24            specificSamplerVect[thread]->
25                sample(state ,genKpar ,dhist ,random);
26
27            //Locate particle in geometry
28            geometry->locate(state);
29        }
30        else{
31            //No sampler specified!
32            printf("pen_specificStateGen:sample: Error: No generic nor specific
33            sampler specified!\n");
34            state.reset();
35            genKpar = ALWAYS_AT_END;
36            dhist = 0;
37        }
38    }
```

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* specifies what thread is calling the sampling method. Remember that specific generators are not thread-safe 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-safe 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. These 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 these functions, tallies can extract the required information from the simulation.

To explain these functions, suppose that we want to create a tally whose 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,

```
1  const unsigned long long nhist
2  const unsigned kdet
3  const pen_KPAR kpar
4  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.

```
1 void pen_tallyDummyLog::tally_beginSim() {
2     //This function is called at the
3     //beginning of the global simulation
4
5     printf("Global simulation begins\n");
6 }
7
```

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.

```
1 void pen_tallyDummyLog::tally_endSim(const unsigned long long nhist){
2
3     //This function is called when the simulation ends,
4     //"nhist" tells us the last history simulated
5
6     printf("Simulation ends at history number %llu\n",nhist);
7 }
8
```

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.

```

1 void pen_tallyDummyLog::tally_beginHist(const unsigned long long nhist,
2                                         const unsigned /*kdet*/,
3                                         const pen_KPAR /*kpar*/,
4                                         const pen_particleState& /*state*/){
5
6     //This function is called when a new history begins
7     printf("Simulation of history %llu begins\n",nhist);
8 }
9

```

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.

```

1 void pen_tallyDummyLog::tally_endHist(const unsigned long long nhist){
2
3     //This function is called when a history ends its simulation
4
5     printf("Simulation of history %llu ends\n",nhist);
6
7 }
8

```

Code 30: Dummy logger *tally_endHist* function.

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

```

1 void pen_tallyDummyLog::tally_beginPart(const unsigned long long nhist,
2                                         const unsigned /*kdet*/,
3                                         const pen_KPAR kpar,
4                                         const pen_particleState& state){
5
6     //Called when the simulation of some particle beggins.
7     //At this point, the particle must be inside the
8     //geometry system into a non void region.
9
10    printf("A %s from histoy %llu begins its simulation.\n",
11           particleName(kpar),nhist);
12    printf("Is located at body %u wich material is %u.\n",
13           state.IBODY,state.MAT);
14
15 }
16

```

Code 31: Dummy logger *tally_beginPart* function.

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

```

1 void pen_tallyDummyLog::tally_endPart(const unsigned long long nhist,
2                                         const pen_KPAR kpar,
3                                         const pen_particleState& /*state*/){
4
5     //This function is called when the
6     //simulation of a particle ends
7
8     printf("A %s from histoy %llu ends its simulation.\n",
9           particleName(kpar),nhist);
10

```

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.

```

1 void pen_tallyDummyLog::tally_move2geo(const unsigned long long /*nhist*/
2                                     ,
3                                     const unsigned /*kdet*/,
4                                     const pen_KPAR kpar,
5                                     const pen_particleState& state,
6                                     const double dsef,
7                                     const double /*dstot*/){
8     //This function is called when a new particle
9     //is sampled into a void zone and, then, is
10    //moved to find a geometry non void zone.
11
12    printf("A %s has been created in a void region.",particleName(kpar));
13    if(state.MAT != 0){
14        printf("After moving it %E cm, has striked the geometry "
15              "body %u which material is %u.\n"
16              ,dsef,state.IBODY,state.MAT);
17    }
18    else{
19        printf("There aren't any non void region on its"
20              "direction (%E,%E,%E).\n",state.U,state.V,state.W);
21        printf("The 'tally_beginPart' function will not be"
22              "triggered for this particle,"
23              "but the 'tally_endPart'.\n");
24    }
25 }

```

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.

```

1 void pen_tallyDummyLog::tally_localEdep(
2     const unsigned long long /*nhist*/,
3     const pen_KPAR kpar,
4     const pen_particleState& /*state*/,
5     const double dE){
6
7     //Called when the particle losses energy locally during its
8     //simulation i.e. the energy loss is not continuous on a
9     //traveled step

```

```

10
11     printf("%s losses energy (%E eV) locally\n",
12     particleName(kpar),dE);
13 }
14

```

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 *dpag* 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 the code 35.

```

1 struct tally_StepData{
2     double dsef;
3     double dstot;
4     double softDE;
5     double softX;
6     double softY;
7     double softZ;
8     unsigned originIBODY;
9     unsigned originMAT;
10 };
11

```

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.

```

1 void pen_tallyDummyLog::tally_step(const unsigned long long /*nhist*/,
2                                   const pen_KPAR kpar,
3                                   const pen_particleState& /*state*/,
4                                   const tally_StepData& stepData){
5
6     //Called after a step call during the particle
7     //simulations i.e. after "tally_beginPart" has been
8     //called for current particle
9
10    if(stepData.dstot > 0.0)
11        printf("%s moves %E cm in the origin material (%d) and "
12        "%E cm in void regions.\n",
13        particleName(kpar),stepData.dsef,
14        stepData.originMAT,stepData.dstot);
15    else
16        printf("%s moves %E cm in the origin material (%d).\n",
17        particleName(kpar),stepData.dsef,
18        stepData.originMAT);
19
20    //Check if the particle losses energy during the step
21    if(stepData.softDE > 0.0){
22        printf("During its travel, the particle losses %E eV.\n",stepData.
23        softDE);
24        printf("Considede that this energy has been "

```



```

24     "deposited at P=(%E,%E,%E).\n",
25     stepData.softX, stepData.softY, stepData.softZ);
26 }
27 }
28

```

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.

```

1 void pen_tallyDummyLog::tally_interfCross(
2     const unsigned long long /*nhist*/,
3     const unsigned /*kdet*/,
4     const pen_KPAR kpar,
5     const pen_particleState& /*state*/){
6     //Called when the particle crosses
7     //an interface during the simulation.
8     printf("%s crossed an interface.\n", particleName(kpar));
9 }
10 }
11

```

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.

```

1 void pen_tallyDummyLog::tally_matChange(
2     const unsigned long long /*nhist*/,
3     const pen_KPAR kpar,
4     const pen_particleState& state,
5     const unsigned prevMat){
6     //Called when the particle crosses an interface and enters
7     //in a new material during the simulation
8
9     printf("%s go from material %u to material %u.\n",
10         particleName(kpar), prevMat, state.MAT);
11 }
12 }
13

```

Code 38: Dummy logger *tally_matChange* function.

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

```

1 void pen_tallyDummyLog::tally_jump(const unsigned long long /*nhist*/,
2     const pen_KPAR kpar,
3     const pen_particleState& state,
4     const double ds){
5     //Called after jump function during the particle simulation
6     printf("%s will try to travel %E cm following "
7         "the direction (%E,%E,%E).\n",
8         particleName(kpar), ds, state.U, state.V, state.W);
9 }
10

```

Code 39: Dummy logger *tally_jump* function.

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

```

1 void pen_tallyDummyLog::tally_knock(const unsigned long long /*nhist*/,
2                                     const pen.KPAR kpar,
3                                     const pen_particleState& /*state*/,
4                                     const int icol){
5     //Called after knock function during the simulation
6     printf("%s has interact with the material via the"
7           " interaction with index %d.\n",
8           particleName(kpar), icol);
9 }
10

```

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 properly without this information.

```

1 void pen_tallyDummyLog::tally_lastHist(const double lasthist){
2     //Called when a source begins its simulation.
3     //"lasthist" tells us what is the initial history
4     //to be simulated at this source.
5
6     printf("New source begins at history number %.0f\n",
7           lasthist);
8
9 }
10

```

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),

```
1
2 pen_EdepMat() : pen_genericTally( USE_ELOSS |
3                               USE_BEGINPART |
4                               USE_BEGINHIST |
5                               USE_ENDHIST |
6                               USE_MOVE2GEO)
7 {}
```

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.

```
1
2 virtual int configure(
3     const wrapper_geometry& /*geometry*/,
4     const abc_material* const /*materials*/[constants::MAXMAT],
5     const pen_parserSection& /*config*/,
6     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.

```

1
2 //Register data to dump
3 dump.toDump(edptmp , nmat) ;
4 dump.toDump(edep , nmat) ;
5 dump.toDump(edep2 , nmat) ;
6 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 code 45.

```

1
2 int createTally(const char* ID,
3               const char* name,
4               const wrapper-geometry& geometry,
5               const abc-material* const materials[constants::MAXMAT],
6               const pen_parserSection& config,
7               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.

4.6 Variance reduction

src/VR

Variance reduction (VR) techniques are implemented in PenRed following the same schema as tallies. That is, each VR technique is implemented as a subclass of their common interface class *pen_genericVR*. This class is a template interface which takes as template argument a particle state type. Like tallies, this class has a set of virtual functions which should be called at different simulation points. However, as VR techniques are intended to change the state of the particle, for example their weight, the particle state is not a constant parameter. Furthermore, as many VR techniques consist on cloning or modifying the particle state, all functions have a variable named *stack* as argument. This one is a standard C++ array of particle states, and is intended to store the particles created by the VR technique. Nevertheless, as the stack could overflow due the use of several VR techniques, the arguments *created* and *available* are used to control the remaining space. The first one, *created*, stores the number of particles stored in the *stack*. For instance, it must be incremented each time the VR technique stores a new state. On the other hand, the *available* variable stores the remaining space in the corresponding stack of secondary particles. So, the remaining space can be calculated as,

$$freeSpace = available - created \quad (7)$$

Similar to particle state generators, we can implement both generic or specific VR techniques. The first ones, generic techniques, can be applied to all particles. For instance, they work with generic particle states i.e. *pen_particleState*. Their source code must be located in **src/VR/generic**. For example, the code 46 shows the header of the splitting class. Notice that both, the register and the specification of used functions is analogous to the tallies procedure, and will be discussed later.

```
1
2 class pen_VRsplitting : public pen_genericVR<pen_particleState>{
3     DECLARE_VR(pen_VRsplitting)
4
5     private:
6
7     // —— splitting numbers, IXRSPL(IBODY).
8     unsigned int ISPL[pen_geoconst::NB];
9     bool LSPL[pen_geoconst::NB];
10
11     double minWght;
12     double maxWght;
13
14     public:
15
16     pen_VRsplitting() : pen_genericVR(VR_USE_INTERFCROSS)
17     {
18         for(unsigned i = 0; i < pen_geoconst::NB; ++i)
19             LSPL[i] = false;
20         minWght = maxWght = 0;
21     }
22
23     int configure(const pen_parserSection& config,
24                 const wrapper_geometry& geometry,
25                 const unsigned verbose);
26
27     void vr_interfCross(const unsigned long long /*nhist*/,
28                       const pen_KPAR /*kpar*/,
29                       const unsigned /*kdet/
```

```

30     pen_particleState& state ,
31     std::array<pen_particleState , constants::NMS>& stack ,
32     unsigned& created ,
33     const unsigned available ,
34     pen_rand& /*random*/ const ;
35 };

```

Code 46: Splitting VR class.

On the other hand, specific VR techniques (located in `src/VR/specific`) use an specific particle state and can't be used on all particles. An example is code 47, where the header of the X-ray splitting VR technique is shown. This one accepts only the photon specific state.

```

1
2 class pen_VRxraysplitting : public pen_genericVR<pen_state_gPol>{
3     DECLARE_VR(pen_VRxraysplitting)
4
5     private:
6
7     // ——— x-ray splitting numbers, IXRSPL(IBODY).
8     unsigned int IXRSPL[pen_geoconst::NB];
9     bool LXRSP[pen_geoconst::NB];
10
11     public:
12
13     pen_VRxraysplitting() : pen_genericVR(VR_USE_PARTICLESTACK)
14     {
15         for(unsigned i = 0; i < pen_geoconst::NB; ++i)
16             LXRSP[i] = false;
17     }
18
19     int configure(const pen_parserSection& config ,
20                 const wrapper_geometry& geometry ,
21                 const unsigned verbose);
22
23     void vr_particleStack(const unsigned long long /*nhist*/,
24                          const pen_KPAR /*kpar*/,
25                          const unsigned /*kdet*/,
26                          pen_state_gPol& state ,
27                          std::array<pen_state_gPol , constants::NMS>& stack ,
28                          unsigned& created ,
29                          const unsigned available ,
30                          pen_rand& random) const ;
31 };

```

Code 47: X-Ray splitting class.

As we can see in the previous codes, the functions to be called during the simulation are constants. Thus, the state of the VR class is unable to change during the simulation. In addition to the *stack*, *created* and *available* arguments, each function gets the history number (*nhist*), the particle type (*kpar*), the particle state (*state*) and the random number generator (*random*). These functions are listed following,

- **vr_particleStack:** This function is intended to be called when a particle is extracted from some stack of secondary particles. The signature of this function is in the code 48,

```

1
2 void vr_particleStack(const unsigned long long nhist ,
3                     const pen_KPAR kpar ,
4                     const unsigned kdet ,

```

```

5     stateType& state ,
6     std::array<stateType, constants::NMS>& stack ,
7     unsigned& created ,
8     const unsigned available ,
9     pen_rand& random) const

```

Code 48: *vr_particleStack* function.

In addition to common arguments, this function takes the detector number where the particle is located (*kdet*). One example is shown in the code 49 which implements the *vr_particleStack* function for the X-Ray splitting class.

```

1
2 void pen_VRxraysplitting::vr_particleStack(const unsigned long long /*
   nhist*/,
3         const pen_KPAR /*kpar*/,
4         const unsigned /*kdet*/,
5         pen_state_gPol& state ,
6         std::array<pen_state_gPol, constants::NMS>& stack ,
7         unsigned& created ,
8         const unsigned available ,
9         pen_rand& random) const{
10
11     if(LXRSPL[state.IBODY] && state.ILB[3] > 0){
12         //Is a characteristic x-ray in a body with x-ray splitting enabled
13         if(state.ILB[0] == 2 && state.ILB[2] < 9){
14             // Unsplitted 2nd generation photon
15             unsigned freeSpace = available - created;
16             unsigned nsplit = std::min(IXRSPL[state.IBODY], freeSpace);
17             state.WGHT /= (double) nsplit;
18             state.ILB[2] = 9; //Labels split x rays
19
20             //Store 'nsplit' states
21             pen_state_gPol stateSplit;
22             stateSplit = state;
23
24             for(unsigned isplit = 1; isplit < nsplit; ++isplit){
25
26                 stateSplit.W = -1.0 + 2.0 * random.rand();
27                 double SDTS = sqrt(1.0 - stateSplit.W*stateSplit.W);
28                 double DF = constants::TWOPI*random.rand();
29                 stateSplit.U = cos(DF)*SDTS;
30                 stateSplit.V = sin(DF)*SDTS;
31                 stack[created++] = stateSplit;
32             }
33         }
34     }
35 }
36 }

```

Code 49: *vr_particleStack* function.

- **vr_interfCross:** The interface cross function is called when the particle crosses an interface. Its signature is the code 50 and takes the same parameters as the *vr_particleStack* function.

```

1
2 void vr_interfCross(const unsigned long long /*nhist*/,
3         const pen_KPAR /*kpar*/,
4         const unsigned /*kdet*/,
5         stateType& /*state*/,
6         std::array<stateType, constants::NMS>& /*stack*/,

```

```

7         unsigned& /*created*/,
8         const unsigned /*available*/,
9         pen_rand& /*random*/) const

```

Code 50: *vr_intfCross* function.

An example of the implementation of this function can be found in the *splitting* class (code 51).

```

1
2 void pen_VRsplitting::vr_intfCross(const unsigned long long /*nhist*/,
3                                   const pen_KPAR /*kpar*/,
4                                   const unsigned /*kdet*/,
5                                   pen_particleState& state,
6                                   std::array<pen_particleState, constants::NMS>& stack,
7                                   unsigned& created,
8                                   const unsigned available,
9                                   pen_rand& /*random*/) const{
10
11     //Check if splitting is enabled in this body
12     if(LSPL[state.IBODY]){
13
14         if(state.WGHT < minWght || state.WGHT >= maxWght)
15             return;
16
17         //Check the available space at the stack
18         unsigned freeSpace = available - created;
19         unsigned nsplit = std::min(ISPL[state.IBODY], freeSpace);
20
21         if(nsplit <= 1)
22             return;
23
24         //Reduce the weight according to splitting factor
25         state.WGHT /= static_cast<double>(nsplit);
26
27         //Clone the state
28         for(unsigned isplit = 1; isplit < nsplit; ++isplit){
29             stack[created++] = state;
30         }
31     }
32 }
33

```

Code 51: Splitting *vr_intfCross* function implementation.

- **vr_matChange**: Material change function is called when the particle cross an interface and change the material. Notice that when *vr_matChange* is called, the function *vr_intfCross* will be called too, because an interface has been crossed. Its signature is code 52. It takes the particle material index before the interface cross as parameter (*prevMat*) in addition to the common variables.

```

1
2 void vr_matChange(const unsigned long long /*nhist*/,
3                  const pen_KPAR /*kpar*/,
4                  const unsigned /*prevMat*/,
5                  stateType& /*state*/,
6                  std::array<stateType, constants::NMS>& /*stack*/,
7                  unsigned& /*created*/,
8                  const unsigned /*available*/,
9                  pen_rand& /*random*/) const

```

Code 52: *vr_matChange* function.

Like tallies, a single VR class doesn't need to implement all the available functions. Instead, the used functions must be specified in the constructor, as shown in the codes 46 and 47. The flags to enable each function are,

VR_USE_PARTICLESTACK

VR_USE_MATCHANGE

VR_USE_INTERFCROSS

and can be combined using the *or* operator. For example, the constructor shown in the code 53 will enable the *vr_particleStack* and the *vr_matChange* functions.

```
1 customVR() : pen_generic(VR_USE_PARTICLESTACK | VR_USE_MATCHANGE)
```

Code 53: VR example constructor class.

In addition to that functions, each VR class must implement their *configure* function, which signature is code 54.

```
1 int configure(const pen_parserSection& config ,
2             const wrapper_geometry& geometry ,
3             const unsigned verbose)
```

Code 54: VR example constructor class.

This one gets the typical section where the parameters specified by the user are stored (*config*), the geometry (*geometry*), and the verbose level (*verbose*) as parameters. Is expected to return 0 if the configuration has been done successfully.

Finally, to register the VR class, we must use the macros,

DECLARE_VR(class-name)

REGISTER_VR(class-name,stateType, ID)

in the same fashion as tallies, sources and other components. Therefore, the *DECLARE_VR* macro must be specified at the beginning of the class definition in the header file, as shown in the codes 46 and 47, taking the name of our VR class as argument. Then, the *REGISTER_VR* macro takes as arguments the class name, the particle type specified in the base class template and an ID to be able to instantiate the class. The last one will be used in the configuration files as *type* value.

Then, the header and source files names must be added to the `src/VR/generic/includes/genericVR.hh` and `src/VR/generic/sources/genericVR.cpp` respectively for generic VR modules, or to `src/VR/specific/includes/specificVR.hh` and `src/VR/generic/specific/specificVR.cpp` for specific VR modules.

4.7 Load Balance

Since version 1.1.0, PenRed incorporates a load balancing system named RUPER-LB (Runtime Unpredictable Performance Load Balancer), a load balancing algorithm for loosely coupled applications running on environments with unpredictable performance variability with both multi-process and multi-thread balance. RUPER-LB is provided as open-source code under the GPLv3 license and can be download as standalone library from <https://github.com/PenRed/RUPER-LB>. A description of the employed algorithm could be found at that repository and at [7].

Using the RUPER-LB nomenclature, each particle source constitute a single task, and each worker represents a thread simulating that source.

In addition, support to use an external server as balance coordinator has been implemented via the ASIO library [8]. As it is in active development and the library interface change between versions, the used version has been included in the PenRed package to avoid compatibility errors.

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 55. 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*.

```

1  class A{
2      protected:
3          int a1, a2;
4      public:
5          double a3;
6
7      A() : a1(0), a2(2), a3(5.3) {}
8
9      void do1();
10     void do2();
11 };
12
13 class B : public A{
14     private:
15 
```

```

16     int b1;
17     public:
18         double b2;
19
20     B() : b1(0), {}
21
22     void doB1();
23     void doB2();
24 };
25

```

Code 55: 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 <i>kpar</i> index (see table 3), only if $ILB[0] > 1$.
ILB[2]	Interaction mechanism <i>ICOL</i> (see tables at section 5.4) that originated the particle, only when $ILB[0] > 1$.
ILB[3]	<p>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,</p> $ILB[3] = Z \times 10^6 + IS1 \times 10^4 + IS2 \times 100 + IS3 \quad (8)$ <p>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 interaction mechanism that caused the initial vacancy in the decaying atom.</p>
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 within the PenRed package. This one allows the user to make simulations without programming. Instead, the user will use a configuration file which structure is explained following.

First, all configuration files must follow the PenRed's internal data library format, which has been explained in section 3. Secondly, all configuration files must include some mandatory sections to specify the simulation characteristics. These sections are, sources, geometry, materials, tallies and global parameters, and all of them are described in the following sections.

6.1 Materials

The material properties, such as cross sections, density, components, etc. are described inside material files. These ones are the same as those used in the original PENELOPE FORTRAN code. Therefore, material files can be generated using the tools and data bases provided at original PENELOPE package, which are also included in the PenRed package. The material data base is located in the folder

`dataBases/penmaterials`

where both, the program code to build materials and the database are stored. That code 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 and PENELOPE. To compile the material build program, the user must compile the “material.cpp” file. For instance, the following line can be used 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 in the same folder where the database is (folder *pdfiles*) and follow the program instructions to create the material. As in the FORTRAN code, a set of predefined materials can be build via numerical codes, which are specified in the appendix A, in the tables 9 and 10.

6.1.1 Configuration

To specify a material in the configuration file, the user must follow the pattern of code 56, where *materials* is a constant text, *material-name* is a text identifier, or name, selected by the user and, finally, *parameter/path* and *value* are the parameters and values to specify for that material.

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

Code 56: Material configuration pattern

For instance, the code 57 shows the copper material configuration used in the first example provided in the package.

```
1 materials/cu/number 1
2
3 materials/cu/eabs_e- 1.0e3
4 materials/cu/eabs_e+ 1.0e3
5 materials/cu/eabs_gamma 1.0e3
6
7 materials/cu/C1 0.05
8 materials/cu/C2 0.05
9
10 materials/cu/WCC 1.0e3
11 materials/cu/WCR 1.0e3
12
13 materials/cu/filename "Cu.mat"
```

Code 57: Complete material configuration example

As is shown, the parameters to specify for each material are analogous to the required by the PENELOPE FORTRAN version. First, the material *number* is used for the geometry material assignation. This one must be greater than 0, thus material 0 is considered as void. Then, *eabs_e-*, *eabs_e+* and *eabs_gamma* specify the electron, positron and gamma absorption energies respectively. The absorption energy units are *eV*. Next, the *C1*, *C2*, *WCC* and *WCR* parameters are used to control the class II transport of electrons and positrons. To be able to select the appropriate parameters, the corresponding description follows, which has been extracted from the PENELOPE manual [2], where are further explained,

- **C1:** Average angular deflection, $C1 \approx 1 - \langle \cos\theta \rangle$, produced by multiple elastic scattering along a path length equal to the mean free path between consecutive hard elastic events. The maximum allowed value is 0.2, but a value of 0.05 is usually adequate.

- **C2**: Maximum average fractional energy loss between consecutive hard elastic events. As *C1*, the maximum allowed value is 0.2 and a value of about 0.05 is, usually, adequate.
- **WCC**: Cutoff energy loss, in eV, for hard inelastic collisions.
- **WCR**: Cutoff energy loss, in eV, for hard bremsstrahlung emission.

The election of these parameters determine the simulation trade off between speed and accuracy. For better accuracy, *C1* and *C2* should have small values (0.01). Larger values makes the simulation faster but less accurate. On the other hand, the cutoff energies *WCC* and *WCR* speed up the simulation when using larger values, but if these are too large, the tallied energy distributions could be distorted. To avoid this effect, the values of *WCC* and *WCR* should be lesser than the bin width used to tally the energy distributions. Finally, to ensure the reliability of the simulation, the user must ensure that the number of steps, or random hinges, per primary track is “statistically sufficient”, which should be achieved with more than 10 steps.

Finally, the *filename* parameter specify the relative path to the material file, which has been previously created with the material builder program. The path to the material file must be enclosed by quotes to be identified as text.

Notice that the configuration shown in the code 57 must be repeated for each used material, changing the corresponding *name* and parameter values.

6.2 Particle sources

Particle sources handle the creation of particles to be simulated. The created particles can be both primary or secondary depending on the selected source. In addition, multiple sources can be specified in the same configuration file. In this case, the sources will be computed sequentially, simulating the first particle of a source after the simulations end of the last particle of the previous source.

To specify a particle source, the user must follow the pattern shown in code 58,

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

Code 58: 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. Via the field *name*, the user specifies a custom name for the particle source. Finally, *parameter/path* specifies the configuration parameter to set with the corresponding *value*. For example, the code 59 shows the basic parameters used by all generic sources type, which name has been set to *source1*.

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

Code 59: Source generic configuration parameters

That source will produce 10^6 primary gamma particles and enables 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 if the particles creation must be restricted to a specific body or material 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 for each source. If specified, when a particle is sampled in a different body or material, 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). Now, we have the generic parameters, but the sampler has not knowledge about how to sampling the initial state of the particle, i.e. position, direction, energy, time etc. Therefore, we need to specify the samplers to use, which are classified in spatial, direction, energy, time and specific samplers. Accordingly, to specify what kind of sampler we are configuring, we will use the keywords *spatial*, *direction*, *energy*, *time* and *specific*. Each sampler type is used with a specific purpose, being spatial, direction, energy and time samplers in charge of determine the particle initial position, direction, energy and live time, respectively. These samplers are cataloged as generic samplers. Nevertheless, specific samplers can set the whole particle state or a portion of it. Therefore, specific samplers can be combined with generic ones depending on the sampler implementation or handle the whole state itself.

Although each sampler has its own parameters, whose can include custom paths, all of them require a parameter named *type*, which sets the sampler to use. For example, to select a mono-energetic energy sampler on the previous source, the line with the required type is shown in code 60. Notice that the source type has been specified after the source name, i.e. with the keyword *energy* and the type value specify which sampler to use among all the available energy samplers.

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

Code 60: Mono-energetic sampler configuration example

In addition, mono-energetic sources only require a single parameter, the particle energy. The next line sets a sampling energy of 30 keV,

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

Code 61: Mono-energetic sampler configuration example

The other samplers are configured analogously, but using their own parameters. To put it all together, a complete generic source configuration is shown in the code 62, i.e. with no specific sampler. In that code, the lines that begins with *#* are considered comments and ignored by the parser.

```
1 #
2 # Source 1
3 #####
4
5 sources/generic/source1/nhist 1.0e6
6 sources/generic/source1/kpar "gamma"
7 sources/generic/source1/record-time true
8
9 # Directional sampling
10 #####
11
12 sources/generic/source1/direction/type "CONE"
13
14
15 # Set theta
16 sources/generic/source1/direction/theta 0.0
17
18 # Set phi
19 sources/generic/source1/direction/phi 0.0
20
```



```

21 # Set oberture (alpha)
22 sources/generic/source1/direction/alpha 5.0
23
24
25 # Energy sampling
26 #####
27
28 sources/generic/source1/energy/type "MONOENERGETIC"
29
30 # Set energy
31 sources/generic/source1/energy/energy 3.0e7
32
33
34 # Spatial sampling
35 #####
36
37 sources/generic/source1/spatial/type "POINT"
38
39 # Set particle origin
40 sources/generic/source1/spatial/position/x 0.0
41 sources/generic/source1/spatial/position/y 0.0
42 sources/generic/source1/spatial/position/z -25.0

```

Code 62: Complete source configuration example

A source without specific sampler, like the previous one, requires at least spatial, energy and direction samplers, 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. In the examples folder we can find configuration files with different sources. In addition, all the available samplers classified by type will be described following.

6.2.1 Spatial source samplers

6.2.1.1 Point

The spatial point sampler needs the (x, y, z) coordinates to fill the position of generation particles. These values are doubles and are specified in cm. Following there is a configuration example of this kind of source sampling corresponding to the 1-disc example.

```

1 # Spatial sampling
2 #####
3
4 sources/generic/source1/spatial/type "POINT"
5
6 # Set particle origin
7 sources/generic/source1/spatial/position/x 0.0
8 sources/generic/source1/spatial/position/y 0.0
9 sources/generic/source1/spatial/position/z -0.0001

```

Code 63: 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. In addition, the configuration requires the size of the box in each axis (dx, dy, dz) in cm, which must be positive or zero. All these values are interpreted as doubles. An example of configuration file is shown below.

```

1 # Spatial sampling
2 #####
3
4 sources/generic/source1/spatial/type "BOX"
5
6
7 # Set box size
8 sources/generic/source1/spatial/size/dx 0.75
9 sources/generic/source1/spatial/size/dy 0.75
10 sources/generic/source1/spatial/size/dz 1.25
11
12 # Set particle origin
13 sources/generic/source1/spatial/position/x 0.0
14 sources/generic/source1/spatial/position/y 0.0
15 sources/generic/source1/spatial/position/z -0.0001

```

Code 64: Spatial source BOX sampler

6.2.2 Direction samplers

6.2.2.1 Cone

Particles generated by the source are sampled in a conical beam. In this sample the overture of the cone must be specified to sample the particles direction uniformly inside the corresponding solid angle. Therefore, the required parameters are the polar (*theta*) and azimuthal (*phi*) angles to determine the solid angle direction, and the semiaperture (*alpha*). Notice that all values are expected in degrees. To reproduce a monodirectional source, the overture must be set to $\alpha = 0.0$. However, to obtain a isotropic source, the semiaperture must be set to $\alpha = 180.0$. All values are expected to be doubles.

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

```

1
2 # Directional sampling
3 #####
4
5 sources/generic/source1/direction/type "CONE"
6
7
8 # Set theta
9 sources/generic/source1/direction/theta 0.0
10
11 # Set phi
12 sources/generic/source1/direction/phi 0.0
13
14 # Set oberture (alpha)
15 sources/generic/source1/direction/alpha 5.0

```

Code 65: Direction CONE sampler

6.2.2.2 Sphere

In this case the direction of the particles generated are sampled in a sphere section. The information needed in this directional sampler is the direction components in each axis (u, v, w), the polar overture introducing the minimum and maximum value of the polar angle (θ_0, θ_1), and the azimuthal overture introducing the minimum value of the phi angle (ϕ_0) and its overture ($d\phi$). All of this values types must be doubles.

6.2.3 Energy samplers

6.2.3.1 Monoenergetic

This energy sampler generates a monoenergetic beam of particles with the specified energy in eV. The only required parameter is the *energy* value, which is interpreted as double and expected to be positive. An example of its usage can be found in the configuration file shown below. These lines correspond to the example 4-x-ray-tube, file `tube.in`.

```
1
2 # Energy sampling
3 #####
4
5 sources/generic/source1/energy/type "MONOENERGETIC"
6
7 # Set energy
8 sources/generic/source1/energy/energy 1.5e5
```

Code 66: Energy samplers MONOENERGETIC

6.2.3.2 Intervals

Intervals sampler generates energies within the specified spectral lines according to the probability assigned to each one. To define the spectral lines in the configuration file, first, the user must specify the number of intervals, *ninterval*, using an integer value. Then, the energy range of each interval, $[lowE, topE]$, are specified as follows. A single array named *lowE* will contain all the low energy boundaries of all energy intervals. In the same way, all values for top energy limits must be grouped in an array named *topE*. Finally, the probabilities of each interval are wrote in a third array named *probabilities*. Therefore, the first position of the array *lowE* specify the low boundary for the first interval or spectral line, the first position of the array *topE* the top limit and the first position of the *probabilities* array the corresponding probability. The same is applied to the second position, which specify the second interval or spectral line, and so on. Notice that the probabilities are not required to be normalised, as is shown in the following example code, which corresponds to the 3-detector example, file `detector1.in`. Regarding the variable types, the energy range values and probabilities are doubles and must be, at least, zero.

```
1
2 # Energy sampling
3 #####
4
5 sources/generic/source1/energy/type "INTERVALS"
6
7
8 # Set intervals
9 sources/generic/source1/energy/nintervals 2
10
11
12 # Set energies
13 sources/generic/source1/energy/lowE [1.17e6,1.33e6]
14 sources/generic/source1/energy/topE [1.17e6,1.33e6]
15
16 # Set probabilities
17 sources/generic/source1/energy/probabilities [50.0,50.0]
```

Code 67: Energy samplers INTERVALS

6.2.3.3 File spectrum

This energy sampler has been implemented to provide compatibility between the PenEasy [9] (v.2020-03-25) energy spectrum format and PenRed. Although this one is described in the PenEasy manual [10], it is also described following for the user convenience.

The format consists on a piecewise function where each entry in the spectrum contains two numbers. The first one, specify the starting energy of a channel, and the second its unnormalized probability. The probabilities must be non-negative and don't require to be normalized to unity. The program interprets a negative probability as the end of the spectrum. For example, a valid spectrum format is shown in the code 68, where the character `#` is interpreted as a comment. Notice that the spectrum file must contain only comments or valid spectrum values until the end of the spectrum. For example, blank lines are not allowed. However, all the text beyond the end of the spectrum is ignored by the sampler.

```
1 #Spectrum      (character '#' is interpreted as a comment)
2 1.20e3 10      1st channel: [1.20,1.61]keV
3 1.61e3 0.0     2nd channel: [1.61,5.00]keV, no emissions
4 5.00e3 20      monoenergetic line at 5.00 keV
5 5.00e3 0       3rd channel: [5.00,6.24]keV, no emissions
6 6.24e3 33.3    4th channel: [6.24,7.32]keV
7 7.32e3 5.02    5th channel: [7.32,8.00]keV
8 8.00e3 10.2    6th channel: [8.00,9.76]keV
9 9.76e3 15.2    7th channel: [9.76,10.0]keV
10 10.0e3 0.0     8th channel: [10.0,15.0]keV, no emissions
11 15.0e3 20      monoenergetic line at 15.0 keV
12 15.0e3 -1      end of the spectrum
```

Code 68: File energy spectrum format

This sampler only requires two configuration values. As the other samplers, the first one is the *type*, which must be set to `"FILE_SPECTRUM"`. The other is the spectrum filename which is introduced as a string, as is shown in the code 69.

```
1
2 # Energy sampling
3 #####
4
5 sources/generic/source1/energy/type "FILE_SPECTRUM"
6 sources/generic/source1/energy/filename "spectrum.spc"
```

Code 69: File spectrum configuration

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.

6.2.5 Specific Sources

6.2.5.1 Phase Space File Source

This type of source uses the phase space file created with the tally type `"PSF"` to run the simulation, from where the particles will be read. An example where this type of source is used is the example 5-accelerator-2, which configuration is shown in the code 70.

Regarding the required parameters, the *filename* is the relative path to of the file from where the particles will be read, and the parameter *E_{max}* specify the maximum energy value of all particles in the PSF. Then, the two next parameters are related with VR techniques. The first one, *wght-window*, sets the interval to apply splitting and Russian Roulette. How both VR techniques are applied is explained following. First, Russian roulette is applied to particles with a weight value lower than the minimum value of the interval. Instead, splitting is applied to particles with a weight value larger than the maximum value of this interval. Therefore, particles with a weight value inside the interval remain unaltered. The second parameter, *split*, sets the number of splits per particle.

In addition to the VR parameters, the *npartitions* parameter is used to set the number of threads used to execute the simulation. The sampler will use this parameter to split the whole PSF in equal regions to be used by different threads. Finally there are two groups of optional parameters to be able to rotate and translate the particles in the PSF file. First, the rotation is specified as a ‘ZYZ’ rotation using the Euler angles *omega*, *theta* and *phi*. Then, the user can apply a displacement using the *dx* parameter for translations in the *x* axis, and *dy* and *dz* for *y*, *z* axes respectively. Notice that the rotation is applied before the translation.

```

1 # Source 1
2 #####
3
4 sources/generic/source-psf/nhist 1.0e6
5 sources/generic/source-psf/specific/type "PSF"
6 sources/generic/source-psf/specific/filename "psf-merged.dat"
7 sources/generic/source-psf/specific/Emax 7e6
8
9 sources/generic/source-psf/specific/wght-window [5e-4, 1e-3]
10 sources/generic/source-psf/specific/nsplit 10
11
12 sources/generic/source-psf/specific/npartitions 6
13
14 sources/generic/source-psf/specific/rotation/omega 0
15 sources/generic/source-psf/specific/rotation/theta 90
16 sources/generic/source-psf/specific/rotation/phi 0
17
18 sources/generic/source-psf/specific/translation/dx 0
19 sources/generic/source-psf/specific/translation/dy 0
20 sources/generic/source-psf/specific/translation/dz 10

```

Code 70: Phase Space File Source

6.3 Geometries

Each simulation only allows to use a single geometry which configuration parameters are specified by the prefix *geometry* to be correctly identified. Like sources, geometries require to specify its type, and other specific parameters.

6.3.1 Quadric

This geometry type is used in all the PENELOPE based examples which have been translated to be reproduced with PenRed. For instance, the code 71 shows the geometry configuration for the first simulation example (1-disc). The geometries for this type are defined in an external file following as is explained in the PENELOPE manual [2] in the Chapter 6 **Constructive quadric geometry**. Moreover, PenRed is compatible with the PENELOPE geometry files, allowing to use them directly from PENELOPE simulations.

```

1 geometry/type "PEN_QUADRIC"
2 geometry/input-file "disc.geo"
3 geometry/processed-geo-file "report.geo"
4
5 geometry/dsmax/1 1.0e-4
6 geometry/kdet/1 1

```

Code 71: Complete geometry configuration example

As other configuration examples, the type is specified by the *type* parameter. Then, the *input-file* parameter specify the relative path to the file where the geometry has been defined. The other parameters *processed-geo-file*, *dsmax* and *kdet* are optional and described following. The first one, *processed-geo-file*, specify a file to generate a geometry report. Secondly, *dsmax* specify the maximum distance allowed for electrons and positrons to jump when class II transport is active. This one is used in thin regions to ensure a minimum number of steps in the specified body. Notice that the body where the *dsmax* is applied is specified via its alias in the parameter path, *geomtry/dsmax/*alias**. For instance, in the previous code, the *dsmax* is applied to the body with the alias "1". Finally, the parameter *kdet* specify the detector identifier which the body belongs. Again, the body is specified using its alias in the parameter path, *geometry/kdet/*alias**.

Optionally, it is possible to specify local absorption energies for any body and particle. In the code 72 we shown an example where the local energy absorption for each particle is set to 10 keV for the body with alias "2". Notice that the most restrictive absorption energy is applied, either the assigned to the corresponding material or the one assigned to the body.

```

1 geometry/eabs/2/electron 1.0e4
2 geometry/eabs/2/gamma 1.0e4
3 geometry/eabs/2/positron 1.0e4

```

Code 72: 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 for each one. 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 in 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 the 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, loads the file to be compared with the original created geometry. In addition, 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 with type and configuration specified by the user, and creates a voxelized geometry file according to that geometry. To do that, the *locate* method is used to map each voxel material and density. As can be found in the mentioned codes, the function to fill a voxel geometry class is named **setVoxels**, and is shown in the code 73, where the initialization of voxels materials and densities is not shown. The required parameters are, a three elements array (*nvox*) with the number of voxels in each axis (*nx, ny, nz*), the voxels sizes in each axis (*sizes*) (*dx, dy, dz*) in cm, a one dimensional array with the material assignation of each voxel (*voxMats*), i.e., with $nx \times ny \times nz$ elements, a one dimensional array with the density factor of each voxel, also with $nx \times ny \times nz$ elements, and a verbose level. Then, once the voxel mesh has been created, the geometry file can be printed in binary format ready to simulate using the function **dump2File**. Also, the voxel geometry information can be printed in ASCII format for visualization purposes using the function **printImage**.

```

1
2 unsigned nvox[3] = ...
3 double sizes[3] = ...
4 unsigned* voxMats = ...
5 double* voxDensFact = ...
6
7 pen_voxelGeo voxelgeo;
8 int err = voxelgeo.setVoxels(nvox, sizes, voxMats, voxDensFact, 3);
9 if(err != 0){
10     printf("Error using 'setVoxels': %d\n", err);
11     return -1;
12 }
13 //Print ASCII file
14 voxelgeo.printImage("voxelGeo.ascii");
15 //Dump binary file
16 voxelgeo.dump2File("voxelGeo.bin");

```

Code 73: Set voxels function and geometry file creation

Regarding the geometry configuration step, the user must specify the parameters (*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 in the code 74.

```

1 geometry/type "VOXEL"
2 geometry/filename "3blocks.vx"
3 geometry/nvoxels/nx 60
4 geometry/nvoxels/ny 60
5 geometry/nvoxels/nz 80
6 geometry/voxel-size/dx 0.2
7 geometry/voxel-size/dy 0.2
8 geometry/voxel-size/dz 0.1
9
10 geometry/dsmax/1 0.05
11 geometry/dsmax/2 0.02
12 geometry/dsmax/3 0.02
13 geometry/dsmax/4 0.50

```

Code 74: 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, which is ready to be simulated directly. As other PenRed components, DICOM geometries require a specific configuration structure to be used. This configuration includes the following parameters,

- **type**: Geometry type, must be set to “DICOM”.
- **directory**: Specifies the relative path where the DICOM images are stored. Notice that all the DICOM images found in 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 in code 75, where we define an interval named *Air*.

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

Code 75: 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 in the corresponding parameters paths. Notice that the contour name must coincide with the contour name stored in the DICOM file. This subsection 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. Contours with lower priority values will be overwritten by contours with higher priority values.

For example, to configure a contour named *target*, the configuration file should contain something like the lines shown in code 76.

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

Code 76: 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 it is required a calibration curve to previously convert HU units to densities. 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 in ASCII format. The data will be stored in a file named *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 in the Figure 9.

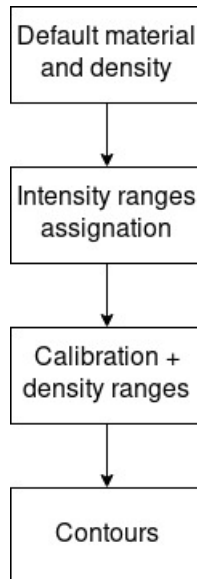


Figure 9: DICOM voxels material an density assign methods hierarchy.

Notice also that the allowed image modalities for DICOM geometry type are, by the moment, Computed Tomography (CT), Ultrasound (US), Radiotherapy Structure Set (RT-STRUCT) and Radio-therapy Plan (RTPLAN). Positron emission tomography (PT) is also accepted, but not for geometry construction purposes. Instead, PET images can be used to create a spatial sampling.

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 [11]. So, the user must download that DICOM to reproduce the example.

6.4 Tallies

Configurations for tallies are similar to sources. The pattern to configure tallies is exemplified in the code 77, where *tallies* is a constant text, *tally-name* is a user defined name for the tally, and *parameter/path* and *value* sets the tally parameters, whose depends on tally type. As sources and geometries, exists several tally types, so the user must specify the parameter *type* on each created tally.

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

Code 77: Tally configuration pattern

Code 78 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.

```
1 tallies/cylDoseDistrib/type "CYLINDRICAL_DOSE_DISTRIB"
2 tallies/cylDoseDistrib/print-xyz true
3 tallies/cylDoseDistrib/rmin 0.0
4 tallies/cylDoseDistrib/rmax 30.0
5 tallies/cylDoseDistrib/nbinsr 60
6 tallies/cylDoseDistrib/zmin 0
7 tallies/cylDoseDistrib/zmax 30.0
8 tallies/cylDoseDistrib/nbinsz 60
```

Code 78: 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.

6.4.1 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 and can be configured to measure the depth absorbed dose distribution. Regarding the units, all results are expressed in (eV cm/g) per history. Below is shown the available parameters of this tally,

- **type “CYLINDRICAL_DOSE_DISTRIB”**: Type name of the tally.
- **print-xyz**: This variable can be set to **true** to print more information in the results file. If activated, two extra values per bin coordinate are printed, which corresponds to the coordinates low and the average values. For *z* coordinates, the average value 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, is set to **false** by default.

- **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.
- **nbinsPhi**: Number of angular (ϕ) bins. By default, this parameter is set to 1.

Regarding the value types, the limits of radial and depth intervals must be doubles, while the number of bins in each case must be integers. The output filename of this tally ends with *cylindricalDoseDistrib.dat* but, if we disable the depth absorbed dose measure in the configuration, the output filename will end with *radialDoseDistrib.dat*.

An example of configuration for this tally is shown in the code 79, which has been extracted from the file *disc.in* of the example 1-disc.

```

1 tallies/cylDoseDistrib/type "CYLINDRICAL_DOSE_DISTRIB"
2 tallies/cylDoseDistrib/print-xyz true (optional)
3 tallies/cylDoseDistrib/rmin 0.0 (mandatory)
4 tallies/cylDoseDistrib/rmax 0.01 (mandatory)
5 tallies/cylDoseDistrib/nbinsr 50 (mandatory)
6 tallies/cylDoseDistrib/zmin 0.0 (mandatory)
7 tallies/cylDoseDistrib/zmax 0.005 (mandatory)
8 tallies/cylDoseDistrib/nbinsz 100 (mandatory)
9 tallies/cylDoseDistrib/nbinsPhi 1 (optional)

```

Code 79: Cylindrical dose distribution tally configuration

6.4.2 Emerging Particles Distribution

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

The 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.

Regarding the values types, the energy limits must be doubles, while the bin numbers must be integers. 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 80 is an example of this tally, which has been extracted from the configuration file *detector1.in* of the example 3-detector-1:

```
1 tallies/EmergingPartDistrib/type "EMERGING_PART_DISTRIB"
2 tallies/EmergingPartDistrib/emin 0.0 (mandatory)
3 tallies/EmergingPartDistrib/emax 1.45e6 (mandatory)
4 tallies/EmergingPartDistrib/nBinsE 280 (mandatory)
5 tallies/EmergingPartDistrib/nBinsTheta 45 (mandatory)
6 tallies/EmergingPartDistrib/nBinsPhi 18 (mandatory)
```

Code 80: Emerging particles tally configuration

6.4.3 Energy Deposition Body

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

The configuration variables of this tally are described following,

- **type "EDEP_BODY"**: Type name of the tally.
- **nBody**: Number of bodies to calculate the deposited energy. All bodies with a index below *nBody* will be tallied.

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

A configuration example of this tally is show in the code 81, which belongs to the file *plane.in* of the example 2-plane.

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

Code 81: Energy deposition in body tally configuration

6.4.4 Energy Deposition Material

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

The parameters required to configure this tally are listed below,

- **type "EDEP_MAT"**: Type name of the tally.
- **nmat**: Number of materials to tally the energy deposition. All materials whith a index lesser than *nmat* will be tallied.

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

The code 82 shows a configuration example obtained from the configuration file *detector1.in* belonging to the example 3-detector-1.

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

Code 82: Energy deposition in material tally configuration

6.4.5 Impact Detector

This tally is capable to measure different magnitudes: fluence spectrum, particle energy spectrum and particle age. Each one is described following,

- **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 corresponding detector number.
- **Energy spectrum:** reports the energy spectrum of particles which enter in the specified detector volume. Particles created inside the detector, for example 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 of the assigned detector.
- **Age:** reports the age distribution of particles which impact to the considered detector, in units of 1/(seconds history). The output file that ends with *age-impdet-num.dat*, contains the probability distribution along each time interval over all simulated particles. As in the previous cases, *num* is the number that identifies the the detector.
- **Energy deposition:** reports the energy deposited spectrum measured in the considered detector. Units are expressed in 1/(eV history). The output file ends with *energyDeposition-impdet-num.dat*, where *num* is the number of the detector. This file contains the probability distribution along each energy interval for the specific detector over all simulated particles.

Regarding the configuration parameters, these are listed below:

- **type “IMPACT_DET”:** Type name of the tally.
- **detector:** Detector index where the measurements of this tally are taken.
- **fluence:** Must be set to **true** to obtain fluence measurements, otherwise set it to **false**. If this information is not specified, **false** value will be assigned by default.
- **emin:** Minimum energy value for tallied particles in fluence, energy spectrum and energy deposition measurements.
- **emax:** Maximum energy value for tallied particles in fluence, energy spectrum and energy deposition measurements. Must be greater than *emin*.
- **nbin-energy:** Number of energy bins, which is used for fluence, energy spectrum and energy deposition. Must be at least 1.
- **spectrum:** Must be set to **true** to obtain spectrum measurements, otherwise set it to **false**. If this information is not specified, **false** is set as default value.
- **age:** Must be set to **true** to obtain age measurements, otherwise set it to **false**. If this information is not specified, **false** is set as default value.
- **enegy-dep:** Must be set to **true** to obtain energy deposition measurements, otherwise set it to **false**. If this information is not specified, **false** value will be assigned by default.

- **linearScale-spc**: Determines if energy spectrum measurements will be tallied using a liner scale (**true**) or a logarithmic scale (**false**). When this parameter is not specified, linear scale is selected by default (**true**).
- **linearScale-edep**: By default is set to **true**, meaning that the energy deposition will be tallied using a linear scale. Instead, if set to **false**, the tally will be created following a logarithmic scale.
- **linearScale-age**: Determines if the output measurements for the age spectrum are expressed using a linear scale (**true**) or a logarithmic scale (**false**). If not specified, a linear scale is selected by default (**true**).
- **nbins-age**: Number of age interval bins. Is only required if **age** is active and must be at least 1.
- **age-min**: Minimum value of the age interval.
- **age-max**: Maximum value of the age interval. Must be greater than *age-min*.

Regarding the parameter types, the interval limits in each case must be doubles and the bin numbers must be specified as integers.

To exemplify the tally usage, next are two examples to show diferent configurations for this tally. First, code 83 shows a part of the configuration file *plane.in* of the 2-plane example, where we the fluence, spectrum and age information are tallied. If fluence, spectrum, age or energy deposition are activated, their corresponding fields are mandatory, otherwise are neither used or expected.

```

1 tallies/ImpactDetector/type "IMPACT_DET"
2 tallies/ImpactDetector/detector 1 (mandatory)
3 tallies/ImpactDetector/fluence true (optional)
4 tallies/ImpactDetector/emin 1.0e5 (mandatory/optional)
5 tallies/ImpactDetector/emax 3.5e7 (mandatory/optional)
6 tallies/ImpactDetector/nbins-energy 100 (mandatory/optional)
7 tallies/ImpactDetector/linearScale-fln true (optional)
8 tallies/ImpactDetector/spectrum true (optional)
9 tallies/ImpactDetector/age true (optional)
10 tallies/ImpactDetector/linearScale-age false (mandatory/optional)
11 tallies/ImpactDetector/nbins-age 100 (mandatory/optional)
12 tallies/ImpactDetector/age-min 1.0e-9 (mandatory/optional)
13 tallies/ImpactDetector/age-max 1.0e-8 (mandatory/optional)

```

Code 83: Impact detector tally configuration example 1

Secondly, code 84 shows some lines of the configuration file *detector1.in* that belongs to the example 3-detector-1. In this example, only the energy deposition information is tallied.

```

1 tallies/ImpactDetector/type "IMPACT_DET"
2 tallies/ImpactDetector/detector 1 (mandatory)
3 tallies/ImpactDetector/emin 0.0e0 (mandatory/optional)
4 tallies/ImpactDetector/emax 1.45e6 (mandatory/optional)
5 tallies/ImpactDetector/nbins-energy 280 (mandatory/optional)
6 tallies/ImpactDetector/energy-dep true (optional)
7 tallies/ImpactDetector/linearScale-edep true (optional)

```

Code 84: Impact detector tally configuration example 2

6.4.6 Spatial Dose Distribution

This tally measures the 3D absorbed dose distribution along the intervals $[xmin, xmax)$, $[ymin, ymax)$, $[zmin, zmax)$ in cm. The units of dose values are eV/g per history. For each coordinates, the user must select the number of bins used to report the data in each axis (nx, ny, nz). In addition, the tally reports the depth dose distribution along the z coordinate in eV/(g/cm²).

The available tally parameters to configure 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 z . Must be greater than $ymin$.
- **nz**: Number of z bins. Must be at least 1.

Considering the parameters type, those who specify coordinate values must be doubles, while the number of bins for each coordinate must be specified as integer. 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. To exemplify the configuration, the code 85 shows an example extracted from the configuration file *disc.in* that belongs to the example 1-disc-novr.

```
1 tallies/SpatialDoseDistrib/type "SPATIAL_DOSE_DISTRIB"
2 tallies/SpatialDoseDistrib/print-xyz true (optional)
3 tallies/SpatialDoseDistrib/xmin 0.0 (mandatory)
4 tallies/SpatialDoseDistrib/xmax 1.0 (mandatory)
5 tallies/SpatialDoseDistrib/nx 1 (mandatory)
6 tallies/SpatialDoseDistrib/ymin 0.0 (mandatory)
7 tallies/SpatialDoseDistrib/ymax 1.0 (mandatory)
8 tallies/SpatialDoseDistrib/ny 1 (mandatory)
9 tallies/SpatialDoseDistrib/zmin 0.0 (mandatory)
10 tallies/SpatialDoseDistrib/zmax 0.005 (mandatory)
11 tallies/SpatialDoseDistrib/nz 100 (mandatory)
```

Code 85: Spatial Dose Distribution tally configuration

6.4.7 Angular Detector

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

Below, we describe the parameters used to configure 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 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.

The expected parameter types are integers for *detector* and the number of bins, and doubles for the energy and angular limits. The output filename for this tally ends with *spc-angdet-num.dat* where *num* is the number of the assigned detector.

The code 86 belongs to the configuration of the example 1-disc-novr, where this tally is used. The configuration file is named *disc.in*.

```

1 tallies/AngularDetector/type "ANGULAR_DET"
2 tallies/AngularDetector/detector 1 (mandatory)
3 tallies/AngularDetector/emin 0.0 (mandatory)
4 tallies/AngularDetector/emax 40.5e3 (mandatory)
5 tallies/AngularDetector/theta1 90.0 (mandatory)
6 tallies/AngularDetector/theta2 180.0 (mandatory)
7 tallies/AngularDetector/phi1 0.0 (mandatory)
8 tallies/AngularDetector/phi2 360.0 (mandatory)
9 tallies/AngularDetector/nBinsE 200 (mandatory)
10 tallies/AngularDetector/linearScale true (optional)

```

Code 86: Angular detector tally configuration

6.4.8 Particle Generation

This tally reports information about the number of 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. In addition, calculates the probabilities for secondary particles to go up bound, down bound and be absorbed.

This tally doesn't require to specify any parameter in 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 code 87. This line is the same for all examples where this tally will be used.

```

1 tallies/secondary/type "SECONDARY_GEN"

```

Code 87: Particle generation tally configuration

6.4.9 Spherical Dose Distribution

This tally reports the absorbed dose distribution, in (eV/g) per history, in a spherical distribution, i.e. the bins follows spherical coordinates. By default, a single bin is used for each angular coordinate and only a radial distribution is tallied in the range $[rmin, rmax)$, specified in cm. The number of radial bins are denoted by *nbin* and, optionally, the 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 configuration parameters are listed following:

- **type “SPHERICAL_DOSE_DISTRIB ”**: Type name of the tally.
- **print-xyz**: Can be 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 this parameter is not specified, will be set to **false** by default.
- **rmin**: Minimum value of radial coordinate. Must be greater than zero.
- **rmax**: Maximum value of radial coordinate. Must be greater than *rmin*.
- **nr**: Number of radial bins. Must be at least 1.
- **ntheta**: Number of polar (θ) bins. By default, this parameter is 1.
- **nphi**: Number of azimuth (ϕ) bins. By default, this parameter is 1.

Regarding the parameter types, all limits are expected to be doubles and the number of bins integers. The output filename ends with *sphericalDoseDistrib.dat*. Any of the provided examples use this tally. However, a configuration example is shown in the code 88.

```
1 tallies/SphericalDose/type "SPHERICAL_DOSE_DISTRIB)"
2 tallies/SphericalDose/print-xyz true (optional)
3 tallies/SphericalDose/rmin 0.0 (mandatory)
4 tallies/SphericalDose/rmax 0.06 (mandatory)
5 tallies/SphericalDose/nr 65 (mandatory)
6 tallies/SphDoseDistrib/ntheta 1 (optional)
7 tallies/SphDoseDistrib/nphi 1 (optional)
```

Code 88: Spherical dose distribution tally configuration

6.4.10 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.

The available tally parameters for the configuration file are listed following:

- **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*.

```

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

```

Code 89: Phase space file tally configuration

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

6.4.11 Kerma track length estimator

This tally is based on the equivalence of particle fluence and the total photon path length per unit volume. A complete description of the estimator can be found at [12]. The estimator could be tallied using three type of meshes: Voxel or cartesian based, cylindrical and spherical meshes. The required configuration for this tally is listed below.

- **type “KERMA_TRACK_LENGTH”**: Type name of the tally.
- **emin**: Photon minimum energy to be considered (eV).
- **emax**: Photon maximum energy to be considered (eV).
- **dataFiles**: One data file with the μ_{en}/ρ (cm^2/g) coefficients per tallied material is required. Each filename is specified following this pattern:

path/to/tally/dataFiles/material-number "filename"

The data files format consists of two columns. The first one stores the energy in eV and the second the corresponding μ_{en}/ρ value in (cm^2/g). The provided points must contain the energy range $[emin, emax]$. However, is not required a constant energy distance between points. Furthermore, there are no limit to the number of points in each data file. Notice that materials with no μ_{en}/ρ data file provided will be ignored by the tally. To obtain the μ_{en}/ρ coefficients, the utility **mutren** can be used, which is provided in the PenRed package. A description can be found in the section 8.1.

- **cartesian**: If this optional section exists, the tally will record the estimator using a regular voxel mesh. The required parameters are listed below:
 - **cartesian/nx**: Number of bins on the X axis.
 - **cartesian/xmin**: Init mesh point on X axis in cm .
 - **cartesian/xmax**: Limit mesh point on X axis in cm .
 - **cartesian/ny**: Number of bins on the Y axis.
 - **cartesian/ymin**: Init mesh point on Y axis in cm .
 - **cartesian/ymax**: Limit mesh point on Y axis in cm .
 - **cartesian/nz**: Number of bins on the Z axis.
 - **cartesian/zmin**: Init mesh point on Z axis in cm .
 - **cartesian/zmax**: Limit mesh point on Z axis in cm .
- **cylindrical**: If this optional section exists, the tally will record the estimator using a cylindrical mesh. The required parameters are listed below:

- **cylindrical/nr**: Number of radial bins.
 - **cylindrical/rmin**: Cylinder minimum radius in *cm*.
 - **cylindrical/rmax**: Cylinder maximum radius in *cm*.
 - **cylindrical/nphi**: Number of angular (ϕ) bins.
 - **cylindrical/nz**: Number of *Z* bins.
 - **cylindrical/zmin**: Init mesh point on *Z* axis in *cm*.
 - **cylindrical/zmax**: Limit mesh point on *Z* axis in *cm*.
- **spherical**: If this optional section exists, the tally will record the estimator using a spherical mesh. The required parameters are listed below:
 - **spherical/nr**: Number of radial bins
 - **spherical/rmin**: Sphere minimum radius in *cm*.
 - **spherical/rmax**: Sphere maximum radius in *cm*.
 - **spherical/ntheta**: Number of polar angular (θ) bins.
 - **spherical/nphi**: Number of azimuth angular (ϕ) bins.

Notice that more than one mesh type could be active at the same tally. In these cases, the tally will generate one report for each mesh type. In code 90 we show a configuration example for that tally.

```

1 tallies/kermaTrackLength/type "KERMA_TRACKLENGTH" (mandatory)
2 tallies/kermaTrackLength/emin 1.0e3 (mandatory)
3 tallies/kermaTrackLength/emax 1.5e6 (mandatory)
4 tallies/kermaTrackLength/dataFiles/1 "mu-Water1000.mat" (mandatory)
5
6 #Optional section:
7 tallies/kermaTrackLength/cylindrical/nr 10 (mandatory)
8 tallies/kermaTrackLength/cylindrical/rmin 0.1 (optional)
9 tallies/kermaTrackLength/cylindrical/rmax 10.0 (mandatory)
10 tallies/kermaTrackLength/cylindrical/nphi 10 (mandatory)
11 tallies/kermaTrackLength/cylindrical/nz 10 (mandatory)
12 tallies/kermaTrackLength/cylindrical/zmin -10.0 (mandatory)
13 tallies/kermaTrackLength/cylindrical/zmax 10.0 (mandatory)
14
15 #Optional section:
16 tallies/kermaTrackLength/cartesian/nx 10 (mandatory)
17 tallies/kermaTrackLength/cartesian/xmin -10.0 (mandatory)
18 tallies/kermaTrackLength/cartesian/xmax 10.0 (mandatory)
19 tallies/kermaTrackLength/cartesian/ny 10 (mandatory)
20 tallies/kermaTrackLength/cartesian/ymin -10.0 (mandatory)
21 tallies/kermaTrackLength/cartesian/ymax 10.0 (mandatory)
22 tallies/kermaTrackLength/cartesian/nz 10 (mandatory)
23 tallies/kermaTrackLength/cartesian/zmin -10.0 (mandatory)
24 tallies/kermaTrackLength/cartesian/zmax 10.0 (mandatory)
25
26 #Optional section:
27 tallies/kermaTrackLength/spherical/nr 10 (mandatory)
28 tallies/kermaTrackLength/spherical/rmin 0.2 (optional)
29 tallies/kermaTrackLength/spherical/rmax 10.0 (mandatory)
30 tallies/kermaTrackLength/spherical/ntheta 10 (mandatory)
31 tallies/kermaTrackLength/spherical/nphi 10 (mandatory)

```

Code 90: Kerma track length tally configuration

Notice that the elements inside an optional section are mandatory only if the section exists. For example, if the section `tallies/kermaTrackLength/cartesian` doesn't exist, any of the cartesian parameters are required, such as `cartesian/nx` or `cartesian/xmin`.

Regarding the parameter types, all bin number are expected to be integers, while the corresponding limits are expected to be doubles.

6.5 Variance Reduction

PenRed implements the very same variance reduction (VR) techniques as the original FORTRAN code and some more. 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. In addition, PenRed implements generic splitting and Russian roulette as VR modules and provide the capabilities to add new ones, as is explained in the section 6.5.

We have two types of VR techniques. First, the context specific techniques, which are implemented inside the particle class, such as interaction forcing or bremsstrahlung splitting. On the other hand, we have the VR techniques implemented as independent modules, such as x-ray and generic splitting or Russian roulette.

6.5.1 Context specific

To use the first kind of techniques, the parameters in the configuration file use the prefix *VR*, as shows code 91. There, *vr-technique* can be interaction forcing (*IForcing*) or bremsstrahlung splitting (*bremss*). Then, on *objectToApply*, the user can select *bodies* or *materials* to apply the variance reduction technique to a single body or to an entire material respectively. Next, *name* is a user defined text identifier for this VR technique. Finally, *parameter/path* and *value* depends on the VR type.

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

Code 91: Variance reduction configuration pattern

First, let's see the interaction forcing (IF) configuration. The code 92 shows a complete interaction forcing configuration used in the example *4-x-ray*. As we can see, this example enables interaction forcing in a single body, which alias is specified by the parameter *body* as a string. Then, specify 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. The next parameter, *factor*, sets the interaction forcing amplification factor and, finally, *min-weight* and *max-weight* limits the weight window where apply this VR technique. Notice that a negative *factor* value will be interpreted as in the PENELOPE main program, i.e., is assumed to mean that a particle with energy $E = E_{MAX}$ should interact, on average, $+factor$ times in the course of its slowing down to rest, for electrons and positrons, or along a mean free path, for photons.

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

Code 92: Interaction forcing configuration for bodies

To set the same interaction forcing on materials, simply substitute *bodies* by *materials*, *body* parameter by *mat-index* and set the material index where apply VR as a integer value. For example, to apply this IF on material 2 the configuration should look like code 93.

```

1 VR/IForcing/materials/VR1/mat-index 2
2 VR/IForcing/materials/VR1/particle "electron"
3 VR/IForcing/materials/VR1/interaction 2
4 VR/IForcing/materials/VR1/factor 400
5 VR/IForcing/materials/VR1/min-weight 0.1
6 VR/IForcing/materials/VR1/max-weight 2

```

Code 93: 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 in the code 94, where *body-alias* must be substituted by the body alias where we want to apply the splitting, *splitting-factor* specify the number of times bremsstrahlung photons will be cloned and *imat* is the material index where apply splitting.

```

1 VR/bremss/bodies/body-alias/splitting splitting-factor
2
3 VR/bremss/materials/mat-index imat
4 VR/bremss/materials/splitting splitting-factor

```

Code 94: Bremsstrahlung splitting configuration for bodies and materials

6.5.2 Specific VR techniques

A specific VR techniques can be used only on a specific particle state type. All of them, when used in the PenRed provided main program, follows the configuration pattern code 95.

```

1 VR/type/vr-name/parameter/path value

```

Code 95: Specific VR technique parameter configuration pattern

Actually, PenRed only provides specific VR techniques for photons. Thus, the only possible value for the specific VR type is *photon*.

```

1 VR/photon/vr-name/parameter/path value

```

Code 96: Photons VR technique parameter configuration pattern

The parameter *vr-name* specify a name for this VR instance and is only used for identification purposes.

6.5.2.1 X-Ray splitting

X-Ray splitting uses a pattern similar to bremsstrahlung case, an example can be found in the example *1-disc-vr* and all the available parameters are summarised in the code 97.

```

1 VR/photon/vr-name/type "XRAY_SPLITTING" (mandatory)
2
3 VR/photon/vr-name/bodies/body-alias/splitting splitting-factor (optional)
4
5 VR/photon/vr-name/materials/mat-index imat (optional)
6 VR/photon/vr-name/materials/splitting splitting-factor (optional)

```

Code 97: X-ray splitting configuration for bodies and materials

First, the VR type is specified by the parameter *type*. Then, the other parameters *VR/photon/name/bodies/...* and *VR/photon/name/materials/...* are analogous to the bremsstrahlung case.

6.5.3 Generic VR techniques

Generic VR techniques can be used on all particle state types. All of them, when used in the PenRed provided main program, follows the configuration pattern shown in the code 98.

```
1 VR/generic/vr-name/parameter/path value
```

Code 98: Specific VR technique parameter configuration pattern

where *vr-name* specify the VR instance name for identification purposes.

6.5.3.1 Splitting

The splitting VR technique consists of cloning the particle and reducing its weight by a factor equal to the number of clones. Thus, if a particle with a weight *WGHT* is split in 10 clones, counting itself, the weight of all the resulting particles will be *WGHT*/10. An example of configuration can be found in the example *8-fake-chamber*. Also, all the parameters are shown in the code 99 and explained below,

- **type**: Specify the VR type, must be set to "*SPLITTING*".
- **minWght**: Specify the minimum particle weight to apply splitting. Particles with lesser weight than the specified will not be split.
- **maxWght**: Specify the maximum particle weight to apply splitting. Particles with greater weight than the specified will not be split.
- **materials/mat-name**: Subsection to specify splitting parameters for materials. Notice that **mat-name** is only used as identifier in the VR instance, and will not be identified with the name used in the geometry configuration. Instead, the material is specified by the integer index **mat-index**. Finally, the number of cloned particles in this material is specified by the **splitting** parameter. This parameter can be repeated for different material indexes.
- **bodies/body-alias/splitting**: There, *body-alias* must identify a valid body in the configured geometry. The specified value of **splitting** will be used in this body. Notice that *bodies* configuration overwrite materials ones. This parameter can be repeated for different bodies.

```
1 VR/generic/vr-name/type "SPLITTING" (mandatory)
2 VR/generic/vr-name/minWght 0.05 (mandatory)
3 VR/generic/vr-name/maxWght 21.0 (mandatory)
4
5 VR/generic/vr-name/materials/mat-name/mat-index 2 (optional)
6 VR/generic/vr-name/materials/mat-name/splitting 20 (optional)
7
8 VR/generic/name/bodies/body-alias/splitting 20 (optional)
```

Code 99: Splitting configuration for bodies and materials

Notice that the splitting is only applied when a particle enter to the material/body after crossing an interface. Therefore, is not applied to secondary particles generated inside the material and is not triggered in bodies if the geometry has not an interface in their boundaries. To force a interface, the user can create a detector or use different materials.

6.5.3.2 Russian roulette

Russian roulette technique kills the particles with a probability specified in the configuration (**prob**). If the particle survive, its weight is multiplied by a factor $1/prob$. All the parameters of the Russian roulette technique are equivalent to the *splitting* class parameters (section 6.5.3.1). The only difference is that the **splitting** parameter is substituted by the **prob** parameter, which specify the particle survival probability. The parameters are summarised in the code 100. In addition, the **type** parameter must be set to "RUSSIAN_ROULETTE" instead of "SPLITTING". An example of configuration can be found in the example *8-fake-chamber*.

```
1 VR/generic/vr-name/type "RUSSIAN_ROULETTE" (mandatory)
2 VR/generic/vr-name/minWght 0.05 (mandatory)
3 VR/generic/vr-name/maxWght 21.0 (mandatory)
4
5 VR/generic/vr-name/materials/mat-name/mat-index 2 (optional)
6 VR/generic/vr-name/materials/mat-name/prob 20 (optional)
7
8 VR/generic/name/bodies/body-alias/prob 20 (optional)
```

Code 100: Russian roulette configuration for bodies and materials

Like in the *splitting* case, Russian roulette only triggers when a particle cross an interface.

6.6 Simulation parameters

Simulation parameters are main specific, unlike source, tally, geometry or material configurations. This section will explain the available parameters for the "pen.main" program. All parameters follows the pattern code 101.

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

Code 101: Simulation parameters configuration pattern

Our main provides 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 can be found in the code 102, 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.

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

Code 102: 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 allows 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 among all specified threads. By default the number of threads is set to one.

```
1 simulation/threads 2
```

Code 103: Number of threads specification

Notice that MPI can be combined with multi-threading. However, the number of MPI processes is not specified in the configuration file. Instead, this 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 in 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, but this feature is intended to be implemented in future PenRed versions.

Following with multi-threading parameters, *thread-affinity* parameter expect a boolean to enable or disable CPU affinity. Actually, this feature can be used only if threads are implemented via *pthread* package, i.e., in most Unix environments.

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

Code 104: Threads affinity

Finally, the user can specify the initial random generator seeds using *seed1* and *seed2* parameters. Another option consists of selecting a seed pair provided by *rand0* [13] 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.

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

Code 105: Initial seeds

6.7 Load balance parameters

As simulation parameters, load balance parameters are main dependent. This optional feature can be enabled during the cmake configuration with the option *WITH_LB*. Actually, the only mandatory parameter to configure the load balance system is *balance-interval* (code 106). This one specify the minimum balance time interval in seconds. Specifying this simple parameter, PenRed simulation threads and MPI processes, if enabled, will be balanced automatically.

In addition, PenRed simulations can be balanced using a centralised server to provide a method to balance simulations across the internet. To start a balance server, the PenRed package provides a balance server example code in `src/utilities/LB/LBserver.cpp`. In addition, in the configuration, the user must specify the balance server hostname or IP as string (`loadBalance/host`) and port (`loadBalance/port`). Also, a worker number identification must be specified (`loadBalance/worker`).

Finally, if the SSL support has been enabled at the cmake configuration with the parameter *WITH_SSL*, the communications between workers and the balance server will be secured with the SSL protocol. This feature requires a set of certificates listed below.

- **CA-cert:** Certification chain file of the trusted CA.
- **cert-file:** Filename of the worker certificate file.
- **key-file:** Filename of the worker key file.
- **key-password:** Worker key password.
- **hostname:** Expected server certificate hostname.

```

1 loadBalance/balance-interval 500      (mandatory)
2 loadBalance/host "server-hostname"   (optional)
3 loadBalance/port 5555                  (mandatory if enabled "host")
4 loadBalance/worker 0                   (mandatory if enabled "host")
5 loadBalance/CA-cert                    (optional)
6 loadBalance/cert-file                  (mandatory if enabled "CA-cert")
7 loadBalance/key-file                   (mandatory if enabled "CA-cert")
8 loadBalance/key-password                (optional)
9 loadBalance/hostname                   (optional)

```

Code 106: Load balance parameters

7 Examples

PenRed has been tested against the original PENELOPE FORTRAN code, on which it is based, using, among other methods, the provided PENELOPE examples. Some parts of the configuration files of these examples have been shown in previous sections to exemplify the different samplers options, source, materials, geometry and tally definitions. At this section, all of the examples included in 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
```

7.1 1-disc

This example presents a point source 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.

7.1.1 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, the impact detector tallies for fluence and energy spectrum and the material energy deposition information.

7.1.2 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.

7.2 2-plane

This example is a semi-infinite water plane with a spherical detector inside. The source is a photon monoenergetic beam of 30 MeV. The detector located in the water plane is defined as an impact detector for fluence measurements. No VR is applied in the plane example. The single material required to execute this example is the *Water.mat* file. The geometry information is read from the *plane.geo* file. The output of this example are the emerging particles distribution, cylindrical and spatial dose distribution, the impact detector tallies corresponding to fluence and energy spectrum measures and output files for material and body energy deposition.

7.3 3-detector

The geometry of this example consists of a NaI cylindrical detector with 5.08 cm of diameter and 5.08 cm height, with 1.27 cm Fe backing. A point-like Co-60 gamma-ray source emits a photon pencil beam in the $-Z$ direction with equiprobable energies 1.17 and 1.33 MeV. The photons impinge on the NaI crystal from above. No VR is applied in this example. Materials required to execute this example are *NaI.mat* and *Fe.mat* files and the geometry is defined in the *detector.geo* file. In this case, the output files correspond to the material and energy deposition and to the emerging particles distribution.

7.4 4-x-ray-tube

The fourth example corresponds to a simple x-ray generator. It consists of a wolframium anode, a filter of aluminium and a silicon detector. The source emits a monoenergetic electron beam of 150 KeV directed to the anode to produce bremsstrahlung photon beam. For this purpose VR techniques are applied to produce splitting in this body. The filter is added to increase the average energy of the resulting beam, and the detector is defined to measure fluence, energy spectrum, and particle age. Materials required for this example are *W.mat*, *Al.mat*, *Si.mat*. The geometry information is in the *tube.geo* file. The output files

are the emerging particles distribution, body and material energy deposition and detector measurements of spectrum, fluence and particle age.

7.5 5-accelerator

The accelerator example simulates a simple electron accelerator and calculates the dose distribution in a water phantom in two steps described following.

7.5.1 5-accelerator-1

In the first step, a phase space file (PSF) is generated at a plane beyond the bottom of the accelerator head, using a planar impact detector. The geometry description of the *accel.geo* file consists of a tungsten target in which impinge an electron beam of 6 MeV, a collimator of the same material and a water phantom. The whole geometry is defined inside an air enclosure. Materials needed to run this simulation are *W.mat*, *H2O.mat* and *Air.mat*. The output files are the body and material energy deposition, the emerging particles distribution and the impact detector spectrum. Moreover, a *psf-merged.dat* file is generated with the PSF information.

7.5.2 5-accelerator-2

In the second step, the initial particle states from that PSF is read and the dose distribution in the water phantom is obtained. The same materials and geometry file are used in this second step. In this case the output files are the emerging particle distribution, body and material energy deposition and the spatial dose distribution in the water phantom.

7.5.3 5-accelerator-3

This example has been created to test the phase space file translation and rotation capabilities. Two variants of this example are created, 5-accelerator-orig and 5-accelerator-rot_trans. The geometry used, *accel.geo*, consists of the same geometry of the previous example 5-accelerator-2 but with some modifications. In this case the water phantom is composed by different water slices with 2 cm of thickness instead of a solid water cube. In the first variation of this example 5-accelerator-orig, the phase space file and the geometry of the problem have not been rotated nor translated. An EDEP_BODY tally type has been set to obtain the deposited energy in each body of the geometry, including the different water slices. Moreover, three impact detector have been created with the tally type IMPACT_DET to obtain fluence and spectrum.

In the second variation, 5-accelerator-rot_trans, the same geometry is used, but a rotation and translation is applied.

- Rotation:
 - omega = 20 degrees
 - theta = 45 degrees
 - phi = 20 degrees
- Translation:
 - y shift = 10 cm

The same rotation and translation have been applied to the phase space file in the config.in file. The same EDEP_BODY and IMPACT_DET tallies used in the first variation have been set

7.6 6-polarisation

This example reproduces the Namito et al.'s (1993) scattering experiment with polarised photons. The source of this example is a monoenergetic polarised photon beam of 40 KeV, defined using the stokes parameters. Interaction forcing is used in this example. The geometry file corresponds to *gpola.geo* and the material files are *Cu.mat* and *Vacuum.mat*. The output files of the execution of this example are the emerging particle distribution, the material energy deposition and the energy spectrum.

7.7 7-aba

This example consist of a ^{60}Co gamma rays source on a three layer cylinder phantom of water-Al-water. The source has equiprobable energies of 1.17 and 1.33 MeV. No VR is applied in this example. The geometry description of the cylinders is defined in the *3discs.geo*, and the material files needed are *water.mat* and *Al.mat*. The output files of this example corresponds to the emerging particles distribution, the impact detector energy deposition, the spatial dose distribution and the material and body energy deposition.

7.8 8-fake-chamber

This example describes an ionization chamber geometry inside a water phantom of $30 \times 30 \times 30 \text{ cm}^3$. The source is a monoenergetic photon beam of 4 MeV. The source-surface distance between the source and the surface of the water phantom is 90 cm. The ionization chamber is located 10 cm from the water surface. The geometry file of this example is the *chamber.geo* and the materials used for its simulation are *air.mat*, *pmma.mat*, *water.mat*. The same files are used for both versions of this example, without and with VR respectively.

7.8.1 8-fake-chamber-novr

In the first version of this example no VR is applied. The output files are the spatial dose distribution and the material energy deposition.

7.8.2 8-fake-chamber-vr

In the second version of this example VR is applied. Splitting and russian roulette are used. The output files are the same that in the first version, the spatial dose distribution and the material energy deposition.

8 Utilities

This section describes the utilities provided with the PenRed package that can be useful for the user. The code of each utility described in this section can be found in the corresponding folder in the directory

`src/utilities/`

8.1 Mutren

The Mutren utility is based on the original `mutren.f` code of the PENELOPE package. This one, calculates the μ_{en} coefficients of the specified material for the specified energies. It consists of a command line program which takes the following parameters as input,

./mutren material-file energy-spectrum-file tolerance sim-time

Each parameter is described following,

- **material-file:** Path to the input material file.
- **energy-spectrum-file:** Path to the energy spectrum file. The format of this file consists of rows where each one contains a single energy in eV.
- **tolerance:** Relative error to stop the simulation, usually set to 0.1%
- **sim-time:** Allowed time to simulate each provided energy in seconds.

The program will calculate the coefficients for each provided energy and store them in a file named **mutren.dat**. Notice that the simulation of each energy point will finish if the objective tolerance has been achieved or if the simulation time reaches the specified limit (**sim-time**).

8.2 iaeaPSF

PenRed writes and reads the PSF in its own binary format. However, the International Atomic Energy Agency (IAEA) provides a standard PSF format. To be able to run IAEA PSF with PenRed, in the utilities folder, user can find *iaeaPSF* folder which includes the IAEA package of routines needed to obtain *iaea2PenRed* and *penRed2iaea* conversion tools. These tools can be found at

src/compiled/iaeaPSF/

The compilation can be set ON or OFF in the ‘compile.sh’ file. On the one hand, the *iaea2PenRed* program allows to convert PSF in IAEA format to PSF in internal binary PenRed format. To do it, user needs both header and phase space file of the IAEA standard format, “IAEAheader” and “IAEAphsp” respectively. To execute this tool the command line needed is

./iaea2PenRed input filename (without extension) and output file name

On the other hand, the *penRed2iaea* program converts a PenRed PSF in to IAEA PSF. In this case the command line to execute it is

./penRed2iaea input filename and output file name (without extension)

providing two new files with extensions “IAEAheader” and “IAEAphsp”, respectively.

8.3 geo2voxel

This utility converts any geometry type in a voxelized one. As program arguments, the **geo2voxel** requires a configuration file which format is the same as **pen_main**, i.e. the format described in the section 3. This configuration file, must include all the configuration parameters to load a geometry of any type, following the descriptions provided in the section 6.3, and the parameters listed below,

- **voxelized/nx:** Number of voxels in the X axis.

- **voxelized/ny**: Number of voxels in the Y axis.
- **voxelized/nz**: Number of voxels in the Z axis.
- **voxelized/dx**: Voxel size in the X axis, in cm.
- **voxelized/dy**: Voxel size in the Y axis, in cm.
- **voxelized/dz**: Voxel size in the Z axis, in cm.
- **voxelized/ox**: Voxelized geometry origin in the X axis, in cm.
- **voxelized/oy**: Voxelized geometry origin in the Y axis, in cm.
- **voxelized/oz**: Voxelized geometry origin in the Z axis, in cm.
- **voxelized/granularity**: Granularity used to determine the material and density of each voxel.

where *granularity* and the number of bins are expected to be integers, and the remaining parameters doubles. The origin of the voxelized geometry (ox, oy, oz) is interpreted as the position of the left bottom corner of the lower Z voxelized geometry plane i.e., is not considered as the center of the first voxel of the mesh. Then, the granularity specify how many points per axis are generated inside each voxel to determine the material and density of each one. The density is obtained from the mean value of all generated points, while the material assignation will be done according to the material index with more points inside the voxel.

An example of configuration to generate a voxelized geometry from a quadric geometry file is shown in the code 107, where 4^3 points are generated in each voxel to determine the material and density and the center of the voxelized mesh has been set to the $(0,0,0)$ of the quadric geometry.

```

1 geometry/type "PEN_QUADRIC"
2 geometry/input-file "disc.geo"
3 geometry/processed-geo-file "report.geo"
4
5 voxelized/nx 300
6 voxelized/ny 300
7 voxelized/nz 120
8
9 voxelized/dx 0.1
10 voxelized/dy 0.1
11 voxelized/dz 0.2
12
13 voxelized/ox -15.0
14 voxelized/oy -15.0
15 voxelized/oz -12.0
16
17 voxelized/granularity 4

```

Code 107: Complete `geo2voxel` configuration example to convert a quadric geometry.

8.4 range

The **range** utility prints the electron, positron and photon ranges for the specified material and energies. The corresponding code is located in the folder **penMats**. The program must be executed via the command line as follows,

```
./range material-file E1 E2 E3
```

where *range* is the executable name, *material-file* is the material file with the material information, the same as the used for simulations, and the following arguments (*E1*, *E2*, *E3*) are a list of energies to calculate the corresponding ranges. Notice that in the execution example we have used 3 energies. However, any number of energies can be used.

8.5 PSF spectrum

The `psf_spectre` utility extracts the energy spectrum distribution of the particles registered in a PSF file, and the corresponding source code can be found in the `psf/psf_spectre.cpp` file. This utility is used via the command line as follows,

```
./psf_spectre PSFfilename emin emax nbins
```

where *psf_spectre* is the utility executable, *PSFfilename* the path to the PSF file to be processed, *emin* the minimum energy, in eV, of the extracted spectrum, *emax* the corresponding spectrum maximum energy and *nbins* the number of spectrum bins.

8.6 PSF to ASCII

The `psf2ascii` utility converts a binary PSF file into a ASCII file format. The source code can be found in `psf/psf2ascii.cpp` and must be used via the command line as follows,

```
./psf2ascii inputFile outputFile
```

where *psf2ascii* is the executable, *inputFile* the binary PSF file path and *outputFile* the filename of the generated ASCII file.

8.7 Registers

The `registers` folder contains a set of sources which compiled programs show all the available types of a specific component. The shown types correspond to the ones used in the configuration files to specify the component *type* parameter. All the available registers are summarised in the Table 8. This ones are executed with no arguments via the command line.

Executable	Registered information
<code>regGenericVR</code>	Generic VR types
<code>regPhotonVR</code>	Specific photon VR types
<code>regGeometries</code>	Geometry types
<code>regSamplers</code>	Particle sampler types
<code>regTallies</code>	Tally types

Table 8: Available register programs with the corresponding registered information.

A Predefined materials

ID	Name	ID	Name	ID	Name
1	Hydrogen	34	Selenium	67	Holmium
2	Helium	35	Bromine	68	Erbium
3	Lithium	36	Krypton	69	Thulium
4	Beryllium	37	Rubidium	70	Ytterbium
5	Boron	38	Strontium	71	Lutetium
6	Amorphous carbon	39	Yttrium	72	Hafnium
7	Nitrogen	40	Zirconium	73	Tantalum
8	Oxygen	41	Niobium	74	Tungsten
9	Fluorine	42	Molybdenum	75	Rhenium
10	Neon	43	Technetium	76	Osmium
11	Sodium	44	Ruthenium	77	Iridium
12	Magnesium	45	Rhodium	78	Platinum
13	Aluminium	46	Palladium	79	Gold
14	Silicon	47	Silver	80	Mercury
15	Phosphorus	48	Cadmium	81	Thallium
16	Sulfur	49	Indium	82	Lead
17	Chlorine	50	Tin	83	Bismuth
18	Argon	51	Antimony	84	Polonium
19	Potassium	52	Tellurium	85	Astatine
20	Calcium	53	Iodine	86	Radon
21	Scandium	54	Xenon	87	Francium
22	Titanium	55	Cesium	88	Radium
23	Vanadium	56	Barium	89	Actinium
24	Chromium	57	Lanthanum	90	Thorium
25	Manganese	58	Cerium	91	Protactinium
26	Iron	59	Praseodymium	92	Uranium
27	Cobalt	60	Neodymium	93	Neptunium
28	Nickel	61	Promethium	94	Plutonium
29	Copper	62	Samarium	95	Americium
30	Zinc	63	Europium	96	Curium
31	Gallium	64	Gadolinium	97	Berkelium
32	Germanium	65	Terbium	98	Californium
33	Arsenic	66	Dysprosium	99	Einsteinium

Table 9: Elements predefined materials extracted from [2]. Material ID corresponds to the element atomic number.

ID	Name
100	Acetone
101	Acetylene
102	Adenine
103	Adipose tissue (ICRP)
104	Air, dry (near sea level)
105	Alanine
106	Aluminum oxide
107	Amber
108	Ammonia
109	Aniline
110	Anthracene
111	B-100 bone-equivalent plastic
112	Bakelite
113	Barium fluoride
114	Barium sulfate
115	Benzene
116	Beryllium oxide
117	Bismuth germanium oxide
118	Blood (ICRP)
119	Bone, compact (ICRU)
120	Bone, cortical (ICRP)
121	Boron carbide
122	Boron oxide
123	Brain (ICRP)
124	Butane
125	N-butyl alcohol
126	C-552 air-equivalent plastic
127	Cadmium telluride
128	Cadmium tungstate
129	Calcium carbonate
130	Calcium fluoride
131	Calcium oxide
132	Calcium sulfate
133	Calcium tungstate
134	Carbon dioxide
135	Carbon tetrachloride
136	Cellulose acetate, cellophane
137	Cellulose acetate butyrate
138	Cellulose nitrate
139	Ceric sulfate dosimeter solution
140	Cesium fluoride
141	Cesium iodide
142	Chlorobenzene
143	Chloroform
144	Concrete, portland
145	Cyclohexane
146	1,2-dichlorobenzene
147	Dichlorodiethyl ether
148	1,2-dichloroethane

149	Diethyl ether
150	N,n-dimethyl formamide
151	Dimethyl sulfoxide
152	Ethane
153	Ethyl alcohol
154	Ethyl cellulose
155	Ethylene
156	Eye lens (ICRP)
157	Ferric oxide
158	Ferroboration
159	Ferrous oxide
160	Ferrous sulfate dosimeter solution
161	Freon-12
162	Freon-12b2
163	Freon-13
164	Freon-13b1
165	Freon-13i1
166	Gadolinium oxysulfide
167	Gallium arsenide
168	Gel in photographic emulsion
169	Pyrex glass
170	Glass, lead
171	Glass, plate
172	Glucose
173	Glutamine
174	Glycerol
175	Graphite
176	Guanine
177	Gypsum, plaster of Paris
178	N-heptane
179	N-hexane
180	Kapton polyimide film
181	Lanthanum oxybromide
182	Lanthanum oxysulfide
183	Lead oxide
184	Lithium amide
185	Lithium carbonate
186	Lithium fluoride
187	Lithium hydride
188	Lithium iodide
189	Lithium oxide
190	Lithium tetraborate
191	Lung (ICRP)
192	M3 wax
193	Magnesium carbonate
194	Magnesium fluoride
195	Magnesium oxide
196	Magnesium tetraborate
197	Mercuric iodide
198	Methane

199	Methanol
200	Mixed wax
201	Ms20 tissue substitute
202	Muscle, skeletal (ICRP)
203	Muscle, striated (ICRU)
204	Muscle-equivalent liquid, with sucrose
205	Muscle-equivalent liquid, without sucrose
206	Naphthalene
207	Nitrobenzene
208	Nitrous oxide
209	Nylon, du Pont elvamide 8062
210	Nylon, type 6 and type 6/6
211	Nylon, type 6/10
212	Nylon, type 11 (rilsan)
213	Octane, liquid
214	Paraffin wax
215	N-pentane
216	Photographic emulsion
217	Plastic scintillator (vinyltoluene based)
218	Plutonium dioxide
219	Polyacrylonitrile
220	Polycarbonate (makrolon, lexan)
221	Polychlorostyrene
222	Polyethylene
223	Polyethylene terephthalate (mylar)
224	Polymethyl methacrilate (lucite, perspex, plexiglass)
225	Polyoxymethylene
226	Polypropylene
227	Polystyrene
228	Polytetrafluoroethylene (teflon)
229	Polytrifluorochloroethylene
230	Polyvinyl acetate
231	Polyvinyl alcohol
232	Polyvinyl butyral
233	Polyvinyl chloride
234	Polyvinylidene chloride (saran)
235	Polyvinylidene fluoride
236	Polyvinyl pyrrolidone
237	Potassium iodide
238	Potassium oxide
239	Propane
240	Propane, liquid
241	N-propyl alcohol
242	Pyridine
243	Rubber, butyl
244	Rubber, natural
245	Rubber, neoprene
246	Silicon dioxide
247	Silver bromide
248	Silver chloride

249	Silver halides in photographic emulsion
250	Silver iodide
251	Skin (ICRP)
252	Sodium carbonate
253	Sodium iodide
254	Sodium monoxide
255	Sodium nitrate
256	Stilbene
257	Sucrose
258	Terphenyl
259	Testes (ICRP)
260	Tetrachloroethylene
261	Thallium chloride
262	Tissue, soft (ICRP)
263	Tissue, soft (ICRU four-component)
264	Tissue-equivalent gas (methane based)
265	Tissue-equivalent gas (propane based)
266	Tissue-equivalent plastic (A-150)
267	Titanium dioxide
268	Toluene
269	Trichloroethylene
270	Triethyl phosphate
271	Tungsten hexafluoride
272	Uranium dicarbide
273	Uranium monocarbide
274	Uranium oxide
275	Urea
276	Valine
277	Viton fluoroelastomer
278	Water, liquid
279	Water vapour
280	Xylene

Table 10: Predefined compounds extracted from [2].

References

- [1] Salvat F., Penelope. a code system for monte carlo simulation of electron and photon transport, Issy-Les-Moulineaux: OECD Nuclear Energy Agency (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. Giménez-Alventosa, G. Moltó, J. D. Segrelles, RUPER-LB: Load balancing embarrassingly parallel applications in unpredictable cloud environments (2020). *arXiv*: 2005.06361.
- [8] ASIO, <https://think-async.com/Asio/>.
- [9] J. Sempau, A. Badal and L. Brualla, A PENELOPE-based system for the automated Monte Carlo simulation of clinacs and voxelized geometries—application to far-from-axis fields, *Med. Phys.* 38 (2011) 5887–5895.
URL <http://dx.doi.org/10.1118/1.3643029>
- [10] J. Sempau, Penelope/peneasy user manual, Tech. rep., Version 2019-01-01.
- [11] 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>
- [12] J. F. Williamson, Monte carlo evaluation of kerma at a point for photon transport problems, *Medical Physics* 14 (4) (1987) 567–576. *arXiv*:<https://aapm.onlinelibrary.wiley.com/doi/pdf/10.1118/1.596069>, doi:10.1118/1.596069.
URL <https://aapm.onlinelibrary.wiley.com/doi/abs/10.1118/1.596069>
- [13] 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>