PenRed: Implementation Manual

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

[1]Departament de Física Atòmica Molecular i Nuclear, Universitat de València , Dr. Moliner, 50, 46100, Burjassot, València, Spain, gimenez.alventosa.vicent@gmail.com
[2]Departament de Física Teòrica and IFIC, Universitat de València-CSIC , Dr. Moliner, 50, 46100, Burjassot, València, Spain,
[3]Instituto de Seguridad Industrial, Radiofísica y Medioambiental (ISIRYM), Universitat Politècnica de València, Camí de Vera s/n, 46022, València, Spain, sanolgi@upvnet.upv.es

May 28, 2023

Abstract

The present document is a manual of the PenRed code system which content is focused on the development of new components and how to include and use them inside the framework. 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, among other features, the capability to read and process DICOM images to perform simulations. These feature facilitate its usage in medical applications. Our framework has been successfully tested against the original PENELOPE Fortran code.

1 Introduction

This document will explain step by step how to implement new components in the PenRed [1] environment. As the intention is to focus exclusively on the development of new components, this document does not provide a deep description of the PenRed structure. However, a deeper description can be found in the PenRed components documentation (in development).

1.1 Document structure

The present document is structured as follows. First, how particle states are defined in PenRed is explained in the section 2. Secondly, the section 3 describes the data structure defined in the framework and its usage. This one is used to provide the necessary information to configure each component. Once the basic components are described, the section 5 contains how to create new geometry modules. The section 6 explains how to implement and include tallies to extract information from the simulation. Then, the modules used to

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

generate new particle states to be simulated follows in the section 7. Finally, the section 8 shows how to implement variance reduction (VR) modules.

2 Particle state

The class `pen_particleState` stores the minimal data required to specify the particles state. The definition of this class can be found in the file,

`src/kernel/states/pen_baseState.hh`

and the variables defined inside the class are described following,

- **E**: Stores the particle energy in eV.
- **X, Y, Z**: Stores the particle position vector (X, Y, Z) respectively, in cm.
- **U, V, W**: Stores the particle **normalized** direction vector (U, V, W).
- **WGHT**: Stores the particle weight, which could be modified, for example, as consequence of variance reduction techniques.
- **IBODY**: Stores the geometry system body index where the particle is located.
- **MAT**: Stores the material where the particle is located.
- **ILB**: Array with 5 components to store particle metadata information, such as the parent particle type. Further details can be found in the *constants and definitions* section in the usage documentation.
- **LAGE**: Enables/disables the particle time recording.
- **PAGE**: Stores the particle life time in seconds.

Although it is possible to define new particle states, like the used for polarised photons (`pen_state_gPol`), notice that all particle states must derive from the provided basic state `pen_particleState`. This restriction ensures the compatibility of any developed state with the PenRed components.

3 Internal data format

To be able to provide a common interface for the input data of all components, regardless the amount of required data, PenRed uses a set of classes to store multiple variables of multiple types in a single instance. These classes can be found in the file,

`src/kernel/parsers/internalData/pen_data.hh`

Among these classes, the `pen_parserSection` class is required by almost all the framework components to get the configuration parameters. To store the data, this class use a basic *key/value* pair format where the key structure is based on the 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. If an array is specified, this one can contain multiple elements of different types.

3.1 Input text format

The users can specify all the required data to configure a components via a text file where each line contains a key and a value separate by white spaces, just like the code 1 shows for the configuration of a cylindrical dose distribution tally.

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

Code 1: Internal data example

Notice that, to be parsed correctly, a string value must be enclosed with double quotes, like "text", as is shown in the previous example. In addition, arrays must be enclosed by [and] and the elements separated by comas. An example extracted from the $7-aba$ quadric example follows,

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

where an array is used to specify the emission probabilities. Notice that an array cannot contain strings, but characters are allowed.

To parse these files, the library includes a function named *parseFile* (code 2). However, the parsing step is usually handled by the main program and is not required to configure a component.

```
int parseFile(const char* filename,
        pen_parserSection& section,
        std::string& errorString,
        long unsigned& errorLine);
```

Code 2: Parse file function

In the code 2, the *filename* parameter specify the name of the file to be parsed, the parameter *section* will be filled with the parsed data, and the *errorString* and *errorLine* parameters will be filled only if an error is produced during the parsing. The first one returns a descriptive message of the parsing error and the second one the line where the error has been found. In addition, to check if the parsing has been done successfully, the returned value of the *parseFile* function is expected to be *INTDATA_SUCCESS*, which is defined in the enumeration `pen_parserErrors`. On error, another value is returned.

3.2 Reading data

To read information from a `pen_parserSection` instance, the *read* method must be used. This one is overloaded for the allowed types, i.e., characters, integers, doubles, bools, `pen_parserData`, strings, `pen_parserArray` and `pen_parserSection`. These functions are shown in the code 3.

```

  //Read functions
```

```
int read(const char* key, char& data, const unsigned index = 0) const

int read(const char* key, int& data, const unsigned index = 0) const

int read(const char* key, double& data, const unsigned index = 0) const

int read(const char* key, bool& data, const unsigned index = 0) const

int read(const char* key, pen_parserData& data, const unsigned index = 0) const

int read(const char* key, std::string& data, const unsigned index = 0) const

int read(const char* key, pen_parserArray& data, const unsigned = 0) const

int read(const char* key, pen_parserSection& data, const unsigned = 0) const
```

Code 3: Parse section read functions

Notice that the *index* parameter is used only to specify the element position when the value to read belongs to an array which key is specified by the parameter *key*. The *data* parameter is used to store the value of the specified key. All read functions returns a *INTDATA_SUCCESS* if the read is done successfully or another value otherwise. An error is returned, for example, if the element specified by the parameter *key* is not convertible to the *data* parameter type.

An example of the *read* function usage is shown in the code 4. There, the key value "energy" is read and stored in the variable *E*.

```
int err;

double E;
err = config.read("energy",E);
if(err != INTDATA_SUCCESS){
  if(verbose > 0){
    printf("monoenergetic:configure:unable to read 'energy' in configuration. Real number expected.\n");
  }
  return -1;
}
```

Code 4: Read example

Notice that it is possible to read a subsection stored in a `pen_parserSection` instance, as the last function in the code 3 suggests. A subsection is interpreted as a "cut" in a key. For example, if we have the following structure in a `pen_parserSection` instance,

```
f1/f2/f3/f4/data1 6
f1/f2/f3/data1 2
f1/f2/f3/data2 4.2
f1/f2/data1 [1,2,3]
f1/data1 1.2
f1/data2 "text"
f1/data3 false
```

and we read the subsection "f1/f2", the resulting `pen_parserSection` instance will contain,

```
f3/f4/data1 6
f3/data1 2
```

```
f3/data2 4.2
data1 [1,2,3]
```

i.e., all keys which prefix do not match with the specified key will not be copied. By default, the prefix is removed also. However, the `readSubsection` function (code 5) can be used for the same purpose but providing the possibility to conserve the prefix key via the parameter *removeKey*.

```
int readSubsection(const char* key, pen_parserSection& secOut, const bool
    removeKey = true)
```

Code 5: Read subsection function

One usage example of this function can be found in the main program. There, the function is used to filter the subsections corresponding to different components, such as geometry, tallies, sources, etc. The code 6 shows an example corresponding to the read of "tallies" section in the main program.

```
  int err = config.readSubsection("tallies",talliesSection);
  if(err != INTDATA_SUCCESS){
    if(verbose > 0){
      printf("createTallies: Error: Configuration 'tallies' section doesn't
    exist.\n");
    }
    return -1;
  }
```

Code 6: Read subsection example

In the previous example, the variable *talliesSection* is a `pen_parserSection` instance.

3.3 Auxiliary functions

In this section some auxiliary functions to handle the PenRed internal data format classes are described.

- **ls**: The **ls** method from the `pen_parserSection` class (code 7) fills a vector of strings (*vect*) with all names in the sections top "folder", i.e. before the first "/". The names are not repeated if several keys shares the same prefix.

  ```
      int ls(std::vector<std::string>& vect)
  ```

  Code 7: Method **ls** from `pen_parserSection` class

3.4 Examples

Several examples of the usage of the internal format classes can be found in the configuration functions of most PenRed components and in the main program. In addition, the tests folder,

```
src/tests/internalData/
```

contains some isolated examples.

4 Automatic image export

PenRed provides a image exporter library to handle different image formats automatically. this one can be found in the folder,

```
src/lib/image
```

That library provides a single structure named *pen_imageExporter* which handles the export process via a user provided function, which must return a specific image pixel value when called. This one must be specified in the constructor. All the available constructors are shown in the code 8. As can be seen, one constructor is defined for each compatible returned variable type. Moreover, two different kind of functions can be provided for the same variable type depending on whether the function returns the associated uncertainty of the value or not. Both will be discussed in the next section.

```cpp
// ** Constructors for images without associated uncertainty
pen_imageExporter(std::function<float(unsigned long long, size_t)> fin);
pen_imageExporter(std::function<double(unsigned long long, size_t)> fin);
pen_imageExporter(std::function<std::int8_t(unsigned long long, size_t)> fin);
pen_imageExporter(std::function<std::uint8_t(unsigned long long, size_t)> fin);
pen_imageExporter(std::function<std::int16_t(unsigned long long, size_t)> fin);
pen_imageExporter(std::function<std::uint16_t(unsigned long long, size_t)> fin);
pen_imageExporter(std::function<std::int32_t(unsigned long long, size_t)> fin);
pen_imageExporter(std::function<std::uint32_t(unsigned long long, size_t)> fin);
pen_imageExporter(std::function<std::int64_t(unsigned long long, size_t)> fin);
pen_imageExporter(std::function<std::uint64_t(unsigned long long, size_t)> fin);

// ** Constructors for images with associated uncertainty
pen_imageExporter(std::function<float(unsigned long long, size_t, float&)> fin);
pen_imageExporter(std::function<double(unsigned long long, size_t, double&)> fin);
pen_imageExporter(std::function<std::int8_t(unsigned long long, size_t, std::int8_t&)> fin);
pen_imageExporter(std::function<std::uint8_t(unsigned long long, size_t, std::uint8_t&)> fin);
pen_imageExporter(std::function<std::int16_t(unsigned long long, size_t, std::int16_t&)> fin);
pen_imageExporter(std::function<std::uint16_t(unsigned long long, size_t, std::uint16_t&)> fin);
pen_imageExporter(std::function<std::int32_t(unsigned long long, size_t, std::int32_t&)> fin);
pen_imageExporter(std::function<std::uint32_t(unsigned long long, size_t, std::uint32_t&)> fin);
pen_imageExporter(std::function<std::int64_t(unsigned long long, size_t, std::int64_t&)> fin);
pen_imageExporter(std::function<std::uint64_t(unsigned long long, size_t, std::uint64_t&)> fin);
```

Code 8: Image exporter constructors

4.1 User provided functions

First of all, the description of the functions with no associated uncertainties follows, as their parameters are common to both function types. The arguments passed to these functions are shown in the example of the code 11, which is used to export the DICOM contour masks in the DICOM based geometry configuration method.

```
    std::function<std::uint8_t(unsigned long long, size_t)> f =
[=, &mask](unsigned long long,
     size_t i) -> std::uint8_t{

  return static_cast<std::uint8_t>(mask[i]);

};

    pen_imageExporter exporter(f);
```

Code 9: Image exporter constructor example for DICOM contour masks

Although the return type can vary, the arguments are the same. The first one, with type *unsigned long long*, is expected to be the number of simulated histories, which is not used in this case. Secondly, the i parameter specify the pixel to be rendered, in the example, it corresponds to the position inside the mask vector. Notice also the cast done to the returned value. As the returned type must match any of the types in the available constructors (code 8), it has been casted from unsigned char to *uint8_t*.

Once constructed, some values must be set in the *pen_imageExporter* instance to export the image properly. First of all, a name must be provided to assign a filename to the exporter image. This is done via the public variable *baseName*, which type is a *std::string*. Then, the number of dimension, number of elements in each dimension and the element size in each dimension must be specified using the member function *setDimensions*. Taking also the DICOM mask exporter example, the number of dimensions in each mask are 3 (x,y,z), and the number of elements and their size coincide with the number and size of voxels in the DICOM, as is shown in the code 11. In addition, a origin could be provided using the method *setOrigin*.

```

  unsigned nElements[3] = {
    static_cast<unsigned>(dicom.getNX()),
    static_cast<unsigned>(dicom.getNY()),
    static_cast<unsigned>(dicom.getNZ())};

  float elementSizes[3] = {
    static_cast<float>(dicom.getDX()),
    static_cast<float>(dicom.getDY()),
    static_cast<float>(dicom.getDZ())};

    exporter.baseName = filename;
    exporter.setDimensions(3,nElements,elementSizes);
    exporter.setOrigin(origin);

    exporter.exportImage(1,pen_imageExporter::formatTypes::MHD);
```

Code 10: Image exporter configuration for DICOM contour masks

Regarding the functions with an associated uncertainties, these ones uses the same arguments as discussed before and another one. That extra parameter is the associated uncertainty to the returned value, which type must match with the returned type. For example, the code

```

```

```
//Register data to create images
unsigned elements[] = {
  static_cast<unsigned>(nx),
  static_cast<unsigned>(ny),
  static_cast<unsigned>(nz)};

float delements[] = {
  static_cast<float>(dx),
  static_cast<float>(dy),
  static_cast<float>(dz)};

// ** Calculate the origin
double origin[3];
// Get geometry offset
geometry.getOffset(origin);
// Add the mesh origin
origin[0] += xmin;
origin[1] += ymin;
origin[2] += zmin;

addImage<double>("spatialDoseDistrib",3,elements,delements,origin,
      [=](unsigned long long nhist,
          size_t i, double& sigma) -> double{

        const double dhists = static_cast<double>(nhist);
        const double fact = ivoxMass[i];
        const double q = edep[i]/dhists;
        sigma = edep2[i]/dhists - q*q;
        if(sigma > 0.0)
          {
      sigma = fact*sqrt(sigma/dhists);
          }
        else
          {
      sigma = 0.0;
          }

        return q*fact;
      });
```

Code 11: Image exporter constructor example for DICOM contour masks

5 Geometry

Geometry modules are used to both, locate and move the particles through the geometry system. In PenRed, all geometry modules must inherit from the `wrapper_geometry` class, defined in the file,

`src/kernel/includes/pen_classes.hh`

Creating a new geometry module directly from the base class `wrapper_geometry` require to define many functions whose behaviour is repeated in most cases, depending on the geometry type. Therefore, PenRed provides two high level classes to develop geometry modules depending on the method to construct the geometry system. The first case, involves geometries constructed via objects, where each object is constituted by a single material. The method to describe these objects is not restricted. For example, the objects can be defined via quadric surfaces, a triangulated mesh etc. In the second case, the geometries are constructed using a mesh where each element gets an independent index and no objects are defined explicitly.

In the next sections the procedure to implement each type of geometry will be explained, but first we will discuss some common information and mandatory methods that must be defined by the developer regardless the geometry type.

5.1 Generic assumptions

To create any new geometry module, this assumptions must be satisfied,

- Any geometry module must be constituted by **bodies**. These bodies are filled with a single **material**, identified by a unsigned integer index. In addition, the **bodies** can belong to a **detector**, which is also identified by a unsigned integer index.
- **Material** index 0 is reserved to void regions.
- **Detector** assignation is optional. If a body does not belong to any detector, the detector index must be set to 0.
- An **interface** is defined as the change of **material** or **detector** index between bodies. Thus, if two bodies in contact shares the same **material** index, there is no interface between both, unless they belong to different **detectors**.
- The particle must be stopped when a **interface** is reached unless the new **material** after the **interface** cross is void. In that case, the particle must be moved across the void region until a non void region is reached or the particle escapes the geometry system.
- A particle escapes from the geometry system when no non void region can be reached.
- When the particle escapes from the geometry system, it is moved an "infinite" distance, usually (10^{35} cm).

5.2 Mandatory functions

The `abc_geometry` and `abc_mesh` classes defines most of the pure virtual methods inherited from the `wrapper_geometry` class for body and mesh based geometries respectively. However, some methods remains to be implemented by the geometry developer in both cases. These methods are listed following,

- **getIBody** (code 12): This method takes a body name (*elementName*) as the only argument. Then, it is intended to return a body index identified by this name. If the name does not corresponds to any defined body, the number of bodies must be returned instead.

```

virtual unsigned getIBody(const char* elementName) const = 0;
```

Code 12: Declaration of the pure virtual method *getIBody* from the `wrapper_geometry` class.

For example, considering the case of quadric geometries (`pen_quadricGeo`), the array *BALIAS* belonging to each body is used to identify the body, as is shown in the code 13.

```

unsigned pen_quadricGeo::getIBody(const char* elementName) const{

  //Construct corrected alias
  char auxAlias[5];
  sprintf(auxAlias,"%4.4s",elementName);
  auxAlias[4] = '\0';
  for(unsigned j = 0; j < getElements(); j++){
    //Check if body alias is the expected one
    if(strcmp(auxAlias,bodies[j].BALIAS) == 0){
      return j;
    }
  }
  return getElements();
}
```

Code 13: Definition of the method *getIBody* from the `pen_quadricGeo` class.

- **locate** (code 14): The *locate* method handles the localization of a particle inside the geometry system. For that purpose, the method takes the particle state as the only parameter. Then, the *locate* method must update the body (*IBODY*) and material (*MAT*) indexes of the input state accordingly to its variables. Usually, only the position (X, Y, Z) is required for this purpose, but all the state is accessible.

```

virtual void locate(pen_particleState& state) const = 0;
```

Code 14: Declaration of the pure virtual method *locate* from the `wrapper_geometry` class.

In the case the particle is not inside the geometry system, the material index must be set to *void* (0) and the body index to a value greater or equal to the number of bodies. An example is shown in the code 15, where the **locate** method for the voxelized geometry class `pen_voxelGeo` is defined.

```

void pen_voxelGeo::locate(pen_particleState& state) const{

  long int ix, iy, iz;

  ix = state.X/dx;
  iy = state.Y/dy;
  iz = state.Z/dz;

  if(ix < 0 || (long unsigned)ix >= nx ||
     iy < 0 || (long unsigned)iy >= ny ||
     iz < 0 || (long unsigned)iz >= nz){
    //Particle scapes from geometry mesh
    state.IBODY = constants::MAXMAT;
    state.MAT = 0;
  }
  else{
    //Particle is in the geometry mesh, calculate voxel
    //index and its material
    long int index = iz*nxy + iy*nx + ix;
    state.MAT = mesh[index].MATER;
    state.IBODY = state.MAT-1;
  }

}
```

Code 15: Definition of the *locate* method for the `pen_voxelGeo` class.

As can be seen, **locate** is defined as a constant method. Thus, is not possible changing the geometry state during that function call.

- **step** (code 16): The **step** method handles the movement of the particle inside the geometry system. To be able to achieve this purpose, all the assumptions of the section 5.1 must be satisfied.

```

virtual void step(pen_particleState& state,
                  double DS,
                  double &DSEF,
                  double &DSTOT,
                  int &NCROSS) const = 0;
```

Code 16: Declaration of the pure virtual method *step* from the `wrapper_geometry` class.

The **step** method takes several parameters, which are listed following,

  - **state**: Provides the particle state to be moved. During the call, the state must be changed to move the particle accordingly. In addition, the body (*IBODY*) and material (*MAT*) indexes must be updated.
  - **DS**: This input parameter stores the maximum distance to be traveled by the particle. The travelled distance can be lesser if a interface is crossed. Notice that the void regions must be skipped even if **DS** is lesser than the required distance to cross a void region.
  - **DSEF**: This output parameter must be filled with the traveled distance in the original material, i.e. in the material index stored in the **state** before moving it. Notice that, if the particle is located in a non void region and a interface with a void region is reached, the distance traveled in the void region after the interface cross must not be scored. However, if the particle is initially located in a void region, **DSEF** must score the traveled distance until the next non void region or until the particle escapes the geometry system.
  - **DSTOT**: This output parameter must be filled with the total traveled distance, regardless if a void region has been crossed or not. Therefore, if the particle does not reached an interface or the material after the interface is not void, the relation $DSTOT = DSEF$ must be accomplished. Instead, if the particle is moving from a non void region and crosses a interface with a void material, the $DSTOT > DSEF$ must be satisfied, because **DSTOT** must score both, the traveled distance in the origin material (**DSEF**) and the traveled distance in the void region.
  - **NCROSS**: This output parameter scores the number of crossed interfaces, i.e. its value must be set to 0 if the particle remains in the original material and detector, or greater than 0 if an interface is crossed.

As the **locate** case, the **step** is defined as a constant method. Thus, it is not possible changing the geometry state during that function call. The code 17 shows the simplest possible **step** function, where the whole space is filled by a single material and no interface can be crossed.

```

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

  DSEF = DS;
  DSTOT = DS;
  NCROSS = 0;
  state.X += DS*state.U;
  state.Y += DS*state.V;
  state.Z += DS*state.W;
}
```

Code 17: Declaration of the *step* method of the `pen_dummyGeo` class.

- **configure** (code 20): The configuration function takes as first argument a `pen_parserSection` (section 3) instance where all the information provided by the user to configure the geometry is included. In addition, the main program includes information about the configured materials. This information is included in a subsection named "materials" where, at this moment, the index and density of each configured material is stored. However this information could be extended in future implementations. The code 18 shows how this information is included.

```
  //Append material information to geometry section

  for(unsigned imat = 0; imat < context.getNMats(); imat++){
    char key[400];
    sprintf(key,"materials/mat%03d/ID",imat+1);
    geometrySection.set(key,(int)imat+1);
    sprintf(key,"materials/mat%03d/density",imat+1);
    geometrySection.set(key,context.readBaseMaterial(imat).readDens());
  }
```

Code 18: Material section included by the main program in the geometry subsection.

An example of how to read the material information can be found in the DICOM geometry configuration function,

`src/geometry/meshes/source/DICOM_geo.cpp`

and is shown in the code 19.

```

  //Read material densities section
  pen_parserSection matSec;
  std::vector<std::string> matNames;
  if(config.readSubsection("materials",matSec) != INTDATA_SUCCESS){
    if(verbose > 0){
      printf("pen_dicomGeo:configure: No material information provided\n"
    );
    }
    configStatus = 4;
    return 4;
  }

  //Extract material names
  matSec.ls(matNames);

  //Iterate over all material
  for(unsigned imat = 0; imat < matNames.size(); imat++){

    double auxDens;
    int auxID;

    std::string idField = matNames[imat] + std::string("/ID");
    std::string densField = matNames[imat] + std::string("/density");

    //Read material ID
    if(matSec.read(idField.c_str(),auxID) != INTDATA_SUCCESS)

    .
    .
    .

    //Read density
    if(matSec.read(densField.c_str(),auxDens) != INTDATA_SUCCESS)

    .
    .
    .
    }
```

Code 19: Geometry material information read in the DICOM configuration function.

The second argument corresponds to a verbosity level, and must be used to control the volume of printed information.

```

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

Code 20: Declaration of the pure virtual method *configure* for the `wrapper_geometry` class.

During the configuration, the variable *configStatus* must be filled with a value that depends on the configuration result. If the geometry has been configured successfully, both, the *configStatus* and the returned value must be set to 0. Instead, a non 0 value on both, the *configStatus* **or** the returned value will be interpreted as a failed configuration.

In addition, depending on the geometry type, some variables must be filled to ensure that the geometry module works properly. These variables will be explained in the sections corresponding to body based (section 5.4) and mesh based (section 5.5) geometries.

5.3 Registering

To be able to use a developed geometry module in the PenRed environment, regardless the kind of geometry, the macros `DECLARE_GEOMETRY` and `REGISTER_GEOMETRY` must be used (code 21).

```

DECLARE_GEOMETRY(Class)
REGISTER_GEOMETRY(Class, ID)
```

Code 21: Provided registration geometry macros.

The parameters of these macros corresponds to the class name (parameter *Class*) and a text identifier to be able to specify the geometry type via the configuration file (parameter *ID*).

The first one, the `DECLARE_GEOMETRY` macro, must be placed inside the class definition, just after the class name, as is shown in the example of the code 22.

```

class pen_quadricGeo : public abc_geometry<pen_quadBody>{
  DECLARE_GEOMETRY(pen_quadricGeo)
   .
   .
  .
}
```
Code 22: Usage of the DECLARE_GEOMETRY in the pen_quadricGeo geometry definition.

The second macro (REGISTER_GEOMETRY), must be placed in the source file, outside the scope of any function, as is shown in the code 23.

```

REGISTER_GEOMETRY(pen_voxelGeo,VOXEL)
```
Code 23: Usage of the REGISTER_GEOMETRY in the pen_voxelGeo geometry source file.

In addition, the header and source files must be included in a specific file depending on the geometry type. This step will be explained in the specific geometry type subsection.

5.4 Body based

Body based geometries inherits from the template abstract class abc_geometry, located in the file,

src/geometry/includes/geometry_classes.hh

Geometries derived from this class must inherit as,

public abc_geometry<bodyType>

where the only template parameter (*bodyType*) specifies the type of the bodies used to construct the geometry. Some implemented examples can be found in the folder,

src/geometry/objects/

5.4.1 Body type

The body type can be defined by the developer to adapt it to the geometry. However, is advisable to define the body type as a derived class of pen_baseBody (code 24), which defines the minimum interface to be compatible with all predefined methods belonging the abc_geometry class.

```

struct pen_baseBody{

  unsigned int MATER;
  unsigned int KDET;
  double DSMAX;
  double localEABS[constants::nParTypes];

  pen_baseBody() : MATER(0), KDET(0), DSMAX(1.0e35){
    //Set body energy cutoffs to "infinite" by default
    for(unsigned i = 0; i < constants::nParTypes; i++){
      localEABS[i] = 1.0e-15;
    }
  }
};
```
Code 24: Definition of pen_baseBody structure.

The required variables for any body are declared in the pen_baseBody class and are described following,

- **MATER:** Material index associated to the body.
- **KDET:** Detector index associated to the body.
- **DSMAX:** Maximum allowed step length for particles with soft energy deposition (electrons and positrons).
- **localEABS:** Array to store the local absorption energies for each particle type. Notice that, in addition, the material absorption energy must be considered to determine the final absorption energy. The most restrictive energy will be used in each body, i.e., if $matEABS$ corresponds to the material absorption energy, the final absorption energy ($EABS$) will be set as,

$$EABS = max(localEABS, matEABS) \tag{1}$$

An example of extended body can be found in the body type defined for the quadric geometry implementation, named `pen_quadBody` (code 25) and located in the file,

`src/geometry/objects/includes/quadric_geo.hh`

```

struct pen_quadBody : public pen_baseBody{

  static const unsigned int NXG = 250;

  char BALIAS[5];

  unsigned int KBODY[NXG];
  unsigned int KBOMO;

  unsigned int  KMOTH;
  unsigned int  KDGHT[NXG];

  pen_bodySurf surfs[NXG];

  .
  .
  .
}
```

Code 25: Definition of `pen_quadBody` structure.

5.4.2 Inherited variables

The `abc_geometry` class provides a few variables that must be used to ensure a proper operation of the developed geometry. These variables are defined following,

- **NB:** Is a constant `unsigned integer` that stores the maximum allowed number of bodies. This constant takes its value from the `pen_geoconst` namespace, which value is 5000 by default. If a different number of allowed bodies is required, the developer must change the value in the `pen_geoconst` namespace. It is located in the file,

  `src/kernel/includes/pen_constants.hh`
- **NBODYS:** Variable `unsigned integer` that stores the number of bodies in the current geometry. It is initialized to 0 and must be updated during the geometry configuration.

- **bodies:** Array of bodies with dimension **NB**. The array type corresponds to the geometry body type, which is determined via the geometry template argument. This array must be filled with the constructed bodies during the geometry configuration. The number of defined bodies in the array is assumed to be equal to **NBODYS**.

5.4.3 Including files

To use a developed body based geometry in the PenRed environment, in addition to the geometry register steps described in the section 5.3, the developer must append a include of the corresponding header and source file names in the files,

```
src/geometry/objects/includes/pen_object_geos.hh
```

and

```
src/geometry/objects/source/pen_object_geos.cpp
```

respectively.

5.5 Mesh based

Mesh based geometries inherits from the template abstract class `abc_mesh`, located in the file,

```
src/geometry/includes/geometry_classes.hh
```

Geometries derived from this class must inherit as,

```
public abc_mesh<meshElement>
```

where the template argument *meshElement* specify the type of the mesh elements used to construct the geometry.

In mesh based geometries, to be able to use all tallies and sources, each non void material is considered as a independent body. Thus, the **IBODY** index can be obtained as,

$$IBODY = MAT \tag{2}$$

for **non void** materials. In addition, the body index 0 is reserved for a enclosure that is supposed to include the whole mesh. This one is created to be able to consider the backscattering effects in the mesh frontiers. Depending on the implementation, the enclosure could be considered as a detector. This approach forces an interface between the enclosure and the mesh regardless the mesh element material. Some implemented examples can be found in the folder,

```
src/geometry/meshes/
```

5.5.1 Mesh element type

The mesh element type can be defined by the developer to fit the mesh geometry requirements. However, it should derive from the provided base type `pen_baseMesh`, which definition is included in the code 26. This base type provides the minimum interface to be compatible with all predefined methods belonging the `abc_mesh` class.

```

struct pen_baseMesh{

  unsigned int MATER;

  pen_baseMesh() : MATER(0){}
};
```

Code 26: Definition of `pen_quadBody` structure.

As is shown, the only requirement for a mesh element is a material index (*MATER*). It is because, as we discussed, mesh based geometries are supposed to identify each non void material with a body. Thus, each element is not a body itself, but a body portion. For this reason, the default mesh element have not a detector index or local energy absorption, as this variables are intended to be set for each body. Moreover, in meshes, it has no sense to define a local energy absorption for each body, because it will produce the same effect as defining the same energy absorption for the whole material.

Therefore, to be able to control the detector index and the maximum distance between hard interactions in soft energy loss steps, a set of variables are defined in the `abc_mesh` class and discussed in the next section.

5.5.2 Inherited variables

The `abc_mesh` class provides some variables that must be used to ensure a proper operation of the developed geometry. These varaibles are defined following,

- **mesh**: Is a pointer which type coincide with the mesh elements type, defined in the template parameter *meshElement*. This pointer will store the geometry mesh as an array of *meshElement*. To set the array size, the developer must use the method *resizeMesh* (code 27), which takes the new mesh dimension as the only parameter. Take into account that the mesh is completely cleaned when is resized, i.e the stored elements are destroyed.

  ```

  void resizeMesh(unsigned dim);
  ```

  Code 27: *resizeMesh* method `pen_quadBody` structure.

  The **mesh** array must be filled during the geometry configuration.

- **meshDim**: This variable stores the dimension of the *mesh* array, i.e. the number of elements that can store. The value of the **meshDim** variable must not be changed, as is handled automatically by the **resizeMesh** method.

- **nElements**: Must store the number of **mesh** elements. The developer must set its value during the geometry configuration.

- **DSMAX**: Array with dimension of the maximum number of allowed materials (*constants::MAXMAT*). This array stores the maximum allowed step length for particles with soft energy deposition (electrons and positrons) in each body. Notice that a body is constituted by all elements with the same non void material. Therefore, the body index can be obtained from the equation 2. This array must be filled during the geometry configuration.

- **KDET**: Array with dimension of the maximum number of allowed materials (*constants::MAXMAT*). Each array element stores the detector index of one body. This array must be filled during the geometry configuration.
- **nBodies**: Variable to store the number of bodies in the geometry, i.e. the number of non void materials plus one for the enclosure. This value must be set during the geometry configuration.

5.5.3 Including files

To use a developed mesh based geometry in the PenRed environment, in addition to the geometry register steps described in the section 5.3, the developer must append a include of the corresponding header and source file names in the files,

```
src/geometry/meshes/includes/pen_mesh_geos.hh
```

and

```
src/geometry/meshes/source/pen_mesh_geos.cpp
```

respectively.

6 Tallies

Tallies are used to extract information from the simulation, such as absorbed energy, particle fluence and spectrums etc. All tallies should be created as derived classes of a common interface defined in the class,

pen_genericTally

which source can be found in,

```
src/tallies/includes/
src/tallies/source/
```

Notice that `pen_genericTally` is a template class which takes the particle state type as unique argument. However, actually, only tallies with the base particle state (class `pen_particleState`) can be used with the provided main program. Therefore, tallies must inherit from,

```
public pen_genericTally<pen_particleState>
```

The following sections describe how to implement a custom tally. To exemplify the procedure, a dummy tally named `pen_tallyDummyLog` has been created and included in the PenRed package. The corresponding files can be used as template to create new tallies,

```
src/tallies/generic/includes/tallyDummyLog.hh
src/tallies/generic/source/tallyDummyLog.cpp
```

In the same folders several examples of different implemented tallies can be found.

6.1 Extracting simulation data

The implementation of tallies, and other components, is done via callbacks defined as virtual functions. These callbacks will be called during the simulation at specific points. Thus, to create a new tally, the developer must know when these callbacks are called an which information is received in each one. Before describing the callbacks, it is advisable to talk about some arguments that are common for most of these functions. These are listed in the code 28,

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

Code 28: Common variables for tally functions.

where *nhist* stores the particle history number, *kdet* the detector where the particle is located, *kpar* the corresponding particle type index and, finally, *state* the actual particle state. The particle indexes can be found in the *Particle indexes* subsection in the *Constants, parameters and definitions* section of the usage documentation. Not all functions get all these parameters as arguments, furthermore, some ones requires another specific parameters. Then, following we will describe all the available callbacks and their parameters. In addition, the code corresponding to the implementation of the tally `pen_tallyDummyLog` is provided for each callback.

Important!

The function **tally_beginHist** has been replaced by **tally_sampledPart** to provide a better support for specific sources. This change applies since version 1.3.1.

- **tally_beginSim**: This function does not get any argument and is called when the global simulation begins, i.e., is not called on every new history. Moreover, is not called when a new source simulation begins. However, notice that if the simulation is resumed via a dump file, this function will be called at the beginning of the resumed simulation too.

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

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

```

Code 29: Dummy logger `tally_beginSim` function.

- **tally_endSim**: Called when the global simulation ends. Thus, this function is expected to be called only one time per simulation. This function takes the number of the last simulated history as the only argument.

```
void pen_tallyDummyLog::tally_endSim(const unsigned long long nhist){

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

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

```

Code 30: Dummy logger `tally_endSim` function.

- **tally_move2geo**: Called when a particle has been sampled by a source into a void region, and has been moved forward to check if it arrives to any non void volume via the *step* geometry function. Notice that the function is called after moving the particle, thus if the particle remains in a void material means that it escaped from the geometry system. Instead, if the particle reaches a non void volume, the simulation will continue, i.e. the particle simulation is considered as begun. Otherwise, this particle simulation never began and will be considered as ended. Therefore, this function is called when the particle simulation has not been started. Notice also that this function is only called when a particle is sampled in a void region. If the particle is sampled in a non void region of the geometry system, this function will not be triggered.

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

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

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

```

Code 31: Dummy logger `tally_move2geo` function.

- **tally_sampledPart**: Called when a new particle is sampled. When this function is called, the state of the sampled particle is the ones which the sampler function creates, i.e. if the particle has been created in a void volume ($MAT = 0$), this function will be called before trying to move the particle to the next non void volume. So, **tally_sampledPart** is triggered before than **tally_move2geo**. The parameter *dhist* corresponds to the number of skipped histories since the previous sampled particle.

Notice that a specific sampler can create several particles within the same history. Moreover, a specific sampler can also skip several histories in a single sample call. For instance, this behaviour is usually found when the phase space file (PSF) sampler is used.

```
void pen_tallyDummyLog::tally_sampledPart(const unsigned long long nhist,
                                        const unsigned long long /*dhist*/,
                         const unsigned /*kdet*/,
                         const pen_KPAR /*kpar*/,
                         const pen_particleState& /*state*/){

    //This function is called when a new particle is sampled
    printf("Simulation of sampled particle in history %llu begins\n",
   nhist);
}

```

Code 32: Dummy logger `tally_sampledPart` function.

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

```
void pen_tallyDummyLog::tally_endHist(const unsigned long long nhist){

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

  printf("Simulation of history %llu ends\n",nhist);

}

```

Code 33: Dummy logger `tally_endHist` function.

- **tally_beginPart**: Called when a particle simulation begins. As explained in the **tally_move2geo** function, a particle simulation begins only if the particle reaches a non void region of the geometry system. Therefore, if the particle is sampled in a void region and remains in a void region after trying to move it forward, this function will not be triggered.

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

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

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

}

```

Code 34: Dummy logger `tally_beginPart` function.

- **tally_endPart**: Called when a particle simulation ends. Unlike the **tally_beginPart** callback, this one will be triggered also when a particle is sampled in a void region and not reaches any non void region of the geometry system.

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

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

```

Code 35: Dummy logger `tally_endPart` function.

- **tally_localEdep**: Called when a particle losses energy locally during its simulation. It is triggered, for example, on interactions or particle absorption, but does not trigger on energy losses by soft interactions during a step. Notice that part of the lost energy can be used to create new particles. So, energy lost is not the energy absorbed by the material. To measure the absorbed energy properly, the energy of the generated secondary particles must be considered in the **tally_beginPart** callback. The argument *dE*, as its name suggest, stores the amount of deposited energy in eV.

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

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

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

```

Code 36: Dummy logger `tally_localEdep` function.

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

  As extra argument, this function receives *stepData*. Its type corresponds to a structure named `tally_StepData` which definition can be found in the code 37.

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

```

Code 37: `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*. That energy has been deposited at the point (*softX,softY,softZ*). Finally, *originIBODY* and *originMAT* save the particle *IBODY* and *MAT* values, respectively, before the *step* call.

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

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

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

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

```

Code 38: Dummy logger `tally_step` function.

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

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

}

```

Code 39: Dummy logger `tally_interfCross` function.

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

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

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

}

```

Code 40: Dummy logger `tally_matChange` function.

- **tally_jump**: Triggered immediately after each *jump* call during the particle simulation. The parameter *ds* stores the distance to travel obtained from the *jump* method.

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

```

Code 41: Dummy logger `tally_jump` function.

- **tally_knock**: Triggered immediately after *knock* call during particle simulation. The parameter *icol* stores the interaction index computed by the *knock* function. The correspondence of these indexes can be found in the PenRed usage manual.

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

```

Code 42: Dummy logger `tally_knock` function.

- **tally_lastHist**: Triggered to update the number of previous histories, for example on source changes or for resumed simulations. Tallies that use information about last registered history, such as the creation of phase space files, can't work property without this information.

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

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

}

```
Code 43: Dummy logger `tally_lastHist` function.

6.2 Callbacks to trigger

Despite the amount of available tally functions, a single tally does not 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, the corresponding flags must be passed as argument in the interface class `pen_genericTally` constructor. These flags are,

USE_BEGINSIM

USE_ENDSIM

USE_SAMPLEDPART

USE_ENDHIST

USE_MOVE2GEO

USE_BEGINPART

USE_ENDPART

USE_JUMP

USE_STEP

USE_INTERFCROSS

USE_MATCHANGE

USE_KNOCK

USE_LOCALEDEP

USE_LASTHIST

where each flag name specify the corresponding function. To provide an example, the code 44 shows the constructor of the material energy deposition tally (`pen_EdepMat` class). This one includes the flags of all necessary callbakcs.

```

pen_EdepMat() : pen_genericTally( USE_ELOSS |
                           USE_BEGINPART |
                           USE_SAMPLEDPART |
                           USE_ENDHIST     |
                           USE_MOVE2GEO)
{}
```
Code 44: Tally flag usage on implemented tallies.

6.3 Mandatory methods

In addition to the optional tally callbacks, 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 in the 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. It is not required to care about the output file names, as the tally cluster will add automatically a prefix to all created files regarding its name, thread identifier and MPI process number. Notice that the *saveData* function can be called only when a history finishes its simulation. The signature is shown in the Code 45.

```
virtual void saveData(const unsigned long long nhist) const = 0;
```

Code 45: Tally **saveData** function.

- **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, all the measured data is expected to be stored in the corresponding final buffers. This function is called automatically when the simulated data is needed, for example before the *saveData* function. The signature is shown in the Code 46.

```
virtual void flush() = 0;
```

Code 46: Tally **flush** function.

- **sumTally**: This function gets a second tally instance of the same type to sum their results. The sum must be stored in the tally which calls *sumTally*, not in the argument instance. On success, this function returns 0. This function is used, for example, to sum-up the results of different threads on multi-threading simulations. The signature is shown in the Code 47.

```
int sumTally(const tallyType& tally);
```

Code 47: Tally **sumTally** function. The *tallyType* type must coincide with the developed tally class type.

- **configure**: As other components, tallies require a configure initialization function. This one takes a configuration structure, a verbose level, a geometry pointer and the material information as arguments, as we can see at the code 48.

```

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

Code 48: Tally configuration function.

  The *config* function must parse and save the user specified parameters to configure the tally. This data must be extracted from the `pen_parserSection` structure, which description can be found in the section 3. The *verbose* parameter 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 virtual mesh elements to calculate the absorbed dose. Notice that the material with index N in the geometry system corresponds to the position $N-1$ in the *materials* array, because the void ($MAT=0$) is not a material itself. Therefore, the void is not included in the *materials* array.

In addition, the configuration function must register the necessary data to store the tally state using the member variable *dump*, which is provided by the tally interface. This *dump*, handles the tally state writing and reading in binary format. Both processes are automatically handled by the tally cluster, however the user must specify which variables must be dumped. To specify the variables to save and load the developer must use the method *toDump*. An example is shown in the code 49.

```

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

Code 49: Tally *dump* registration for `tallyEnergyDepositionMat`.

As we can see, only one function is required to register the tally data. The *toDump* function 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 to register from that pointer. Notice that *toDump* is overloaded to accept the basic data types in order to simplify its usage. However, can't handle user defined structures.

6.4 Optional non callback methods

In addition to mandatory methods, there exist some implementable methods to provide access to some specific information and features. These ones are optional and can be omitted if are not required in our tally.

- **sharedConfig**: This function gets a second tally instance of the same type to get values after the configuration or perform any other task. The function is called just after the configuration function. It is only called by tallies with a thread identifier greater than 0 getting as argument the corresponding tally with the thread identifier 0. Therefore, this function exists to be able to perform expensive configuration calculus in the thread 0 tally and, then, share the results with the other threads, avoiding repeating the calculus and duplication of data. The function must return a value of 0 on success. The signature is shown in the Code 52.

```
int sharedConfig(const tallyType& tally)
```

Code 50: Tally **sharedConfig** function. Notice that the *tallyType* type must coincide with the developed tally class type.

6.5 Auxiliary functions

Following are already implemented and available helper functions that can be used to develop a tally,

- **getThread**: Returns the thread ID of this tally instance.

```
inline unsigned getThread() const
```
Code 51: Tally **getThread** function.

- **readStack**: Gets a *kpar* value as argument and returns the particle stack corresponding to this *kpar*. Notice that the stack is only available for reading and its state can not be modified. Notice that stacks are only available in the simulation callbacks and can not be used in the configuration function.

```
inline const abc_particleStack* readStack(const pen_KPAR kpar) const
```
Code 52: Tally **readStack** function.

6.6 Register

Implemented tallies must be registered using the

DECLARE_TALLY

and

REGISTER_COMMON_TALLY

macros (code 53). These macros, takes as arguments the class name of the developed tally (*Class* parameter), the particle state type required by the tally (*State* parameter) and a unique tally name (*ID* parameter), which will be used in the configuration file to specify its type.

```

 DECLARE_TALLY(Class,State)
 REGISTER_COMMON_TALLY(Class, ID)
```
Code 53: Tally register macros.

The first one must be placed inside the class declaration, just after the class name, as is shown in the code 54. That code corresponds to the `pen_tallyDummyLog` class definition.

```

class pen_tallyDummyLog : public pen_genericTally<pen_particleState> {
 DECLARE_TALLY(pen_tallyDummyLog,pen_particleState)
.
.
.
}
```
Code 54: Tally `pen_tallyDummyLog` declaration macro.

The second one must be placed in the source file, as is shown in the code 55.

```

REGISTER_COMMON_TALLY(pen_tallyDummyLog, DUMMY_LOG_EXAMPLE)
```
Code 55: Tally `pen_tallyDummyLog` register macro.

Finally, a include of the header and source file names must be appended to the lists inside the files

`src/tallies/generic/includes/genericTallies.hh` and

```
src/tallies/generic/source/genericTallies.cpp
```

respectively.

7 Source samplers

Source samplers are used to sample the particle states to be simulated. The samplers are classified in five categories, `spatial`, `direction`, `energy`, `time` and `specific`. Each one has its own purpose, which is explained following,

- **Spatial**: Samples initial particle position X,Y,Z. The $IBODY$ and MAT identifiers will be determined using the sampled position, but are not set during the sampling.
- **Direction**: Samples initial particle direction vector (U,V,W).
- **Energy**: Samples initial particle energy E.
- **Time**: Samples initial particle age (variable $PAGE$). This is an **optional** sampler type.
- **Specific**: Unlike previous samplers, specific samplers gets the whole particle state and can modify all its variables. In addition, specific samplers are not restricted to be used on generic particle states, and can be defined for other states.

Thus, we will consider the **Spatial**, **Direction**, **Energy** and **Time** samplers as generic samplers, because these can be used in any particle state.

How to implement a sampler of each type will be described in the following sections. The main differences between different samplers are the state variables which are calculated by the sampler. However, all samplers shares a common function that must be implemented by the developer. This one is the *configure* virtual method, which signature is sampler dependent. However, all *configure* methods must return an integer value. A returned 0 value is interpreted as success, while nonzero as failed.

In addition, the samplers could require geometry information to perform their calculus. For this reason, when a geometry is set to the samplers, the virtual method *updateGeometry* (code 56) is triggered and the sampler can use and store any information provided by the geometry. This method can be used in all sampler types and must be overwritten by the developer to use the geometry information, if it is required. Notice that during the *configure* execution the geometry is not accessible, thus must be accessed in the *updateGeometry* method.

```

virtual void updateGeometry(const wrapper_geometry* geometryIn){}
```

Code 56: Sampler `updateGeometry` virtual function.

7.1 Register

To register a sampler, the developer must use the provided macros to declare and register the sampler. On the one hand, the “declare sampler” macro is shown in the code 57, which gets a single argument corresponding to the sampler class name.

```

DECLARE_SAMPLER(Class)
```

Code 57: Sampler declaration macro.

This macro must be used inside the sampler class definition, just after the class name, as is shown in the code 58 for the spatial box sampler case. The same procedure is used to declare all samplers, regardless its type.

```

class box_spatialSampling : public abc_spatialSampler {

  DECLARE_SAMPLER(box_spatialSampling)

   .
   .
   .
}
```

Code 58: Sampler declaration macro example.

On the other hand, two “register sampler” macros are provided, which are shown in the code 59.

```

REGISTER_SAMPLER(Class, ID)
REGISTER_SPECIFIC_SAMPLER(Class, State, ID)
```

Code 59: Sampler registration macros.

Only specific samplers must use the **REGISTER_SPECIFIC_SAMPLER** macro, while the other types must use the **REGISTER_SAMPLER** macro. There, the *Class* parameter corresponds to the sampler class name, the specific sampler compatible particle state class is specified by the *State* parameter, and the *ID* parameter provides a unique identifier for the registered sampler. Both macros must be called in the source file, out of any function scope. An example of each macro usage is shown in the code 60, where the *box_spatialSampling* and *gammaPolarised_specificSampler* are used for the generic and specific sampler registration respectively.

```

REGISTER_SAMPLER(box_spatialSampling,BOX)
REGISTER_SPECIFIC_SAMPLER(gammaPolarised_specificSampler,pen_state_gPol,
    GAMMA_POL)
```

Code 60: Sampler register macros.

However, notice that a specific sampler does not require to use a non generic particle state. In fact, a specific sampler can be used to sample basic particle states. This is the case of the phase space file sampler, which register macro is shown in the code 61.

```

REGISTER_SPECIFIC_SAMPLER(psf_specificSampler,pen_particleState, PSF)
```

Code 61: Sampler register macro with generic particle state.

7.2 Spatial

Custom `spatial` samplers must inherit from the base class `abc_spatialSampler` as,

```
public abc_spatialSampler
```

This base class is located in the file,

`src/particleGen/includes/pen_samplers.hh`

and several examples of `spatial` sampler implementations can be found in,

`src/particleGen/spatial`

The samplers are intended to provide a initial position to be rotated and translated. Thus, the resulting sampled position (X'), **in cm**, follows the expression,

$$X' = T + R \cdot X, \tag{3}$$

where T is a translation, R a rotation matrix, and X the initial sampled position. Both, T and R are variables provided in the base class `abc_spatialSampler`, corresponding to the

```
double translation[3];
```

and

```
double rotation[9];
```

variables, respectively. On the one hand, the *translation* variable can be accessed directly in derived classes and is intended to be filled in the *configure* function. On the other hand, the *rotation* variable should be filled also during the *configure* execution, but using the public method *setRotationZYZ* (code 62), which takes as arguments three Euler angles to calculate the final rotation matrix. Notice that the Euler angles axis follows the same schema as the PENELOPE package, i.e. the rotations are performed using the Z, Y, Z axis. Notice that the rotation is optional, thus if no rotation matrix constructed using the *setRotationZYZ* method, no rotation will be applied to the sampled position.

```

void setRotationZYZ(const double omega, const double theta, const double phi)
```

Code 62: Set rotation method.

There are no more common variables to fill in the spatial sampler, remaining the variables required by the implemented sampler. Turning on functions, the only two required methods to implement a `spatial` sampler are the *configure* and the *geoSampling* methods, which signature is shown in the code 63.

```

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

  virtual void geoSampling(double pos[3], pen_rand& random) const = 0;
```

Code 63: Spatial sampler mandatory functions.

First, the *configure* method takes a `pen_parserSection` (section 3) as first argument, which includes all the data specified by the user to configure the sampler. In addition, the parameter *verbose* must be used to control the number of prints. An example is shown in the code 64, which corresponds to the box spatial sampler. As can be seen in the configuration function, this one does not use rotations.

```

```

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

  int err;

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

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

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

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

  dx05 = dx*0.5;
  dy05 = dy*0.5;
  dz05 = dz*0.5;

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

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

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

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

  return 0;
}
```

Code 64: `box_spatialSampling` configure function.

Secondly, the *geoSampling* method gets an array of doubles with dimension three, which must be filled with the sampled x, y, z position (vector X from equation 3). Also, a random number generator is provided via the *random* parameter. Again, the box spatial sampler is used as example and the corresponding *geoSampling* method is shown in the code 65.

```

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

  pos[0] = dx*random.rand()-dx05;
  pos[1] = dy*random.rand()-dy05;
  pos[2] = dz*random.rand()-dz05;

}
```

Code 65: Spatial sampler *geoSampling* function.

Notice that the *geoSampling* method is defined as constant. Therefore, this method can't change the sampler state. This is only allowed in the *configure* and *updateGeometry* calls.

7.2.1 Including files

To use a developed spatial sampler in the PenRed environment, in addition to the sampler register steps described in the section 7.1, the developer must append a include of the corresponding header and source file names in the files,

`src/particleGen/spatial/includes/spatialSamplers.hh`

and

`src/particleGen/spatial/source/spatialSamplers.cpp`

respectively.

7.3 Direction

`Direction` samplers must inherit from the base class `abc_directionSampler` as,

`public abc_directionSampler`

The base class is located in the file,

`src/particleGen/includes/pen_samplers.hh`

and several examples of **direction** sampler implementations can be found in,

`src/particleGen/direction`

This kind of samplers are intended to provide the initial **normalized** direction (U, V, W) of the particle. To achieve this purpose, the direction samplers do not require to fill any variable of the base class, unlike the spatial sampling case. Therefore, a direction sampler only requires to implement the *configure* and the sampling function, named *directionSampling*, which signatures are shown in the code 66.

```cpp

virtual int configure(const pen_parserSection&, const unsigned = 0) = 0;
virtual void directionSampling(double dir[3], pen_rand& random) const = 0;
```

Code 66: Direction sampler mandatory functions.

First, the *configure* method takes a `pen_parserSection` (section 3) as first argument, which includes all the data specified by the user to configure the sampler. In addition, the parameter *verbose* must be used to control the number of prints. An example is shown in the code 67, which corresponds to the conic direction sampler.

```cpp

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

  int err;
  double theta,phi,alpha;
  //Store cosines (u,v,w)
  err = config.read("theta",theta);
  if(err != INTDATA_SUCCESS){
    if(verbose > 0){
      printf("coneDirection:configure:unable to read 'theta' in configuration. Real number expected\n");
    }
    return -1;
  }
  err = config.read("phi",phi);
  if(err != INTDATA_SUCCESS){
    if(verbose > 0){
      printf("coneDirection:configure:unable to read 'phi' in configuration. Real number expected\n");
    }
    return -1;
  }
  err = config.read("alpha",alpha);
  if(err != INTDATA_SUCCESS){
    if(verbose > 0){
      printf("coneDirection:configure:unable to read 'alpha' in configuration. Real number expected\n");
    }
    return -1;
  }

  if(verbose > 1){
    printf("Theta: %12.4E DEG\n",theta);
    printf("Phi  : %12.4E DEG\n",phi);
    printf("Alpha: %12.4E DEG\n",alpha);
  }

  theta *= deg2rad;
  phi   *= deg2rad;
  alpha *= deg2rad;

  CPCT  = cos(phi)*cos(theta);
  CPST  = cos(phi)*sin(theta);
  SPCT  = sin(phi)*cos(theta);
  SPST  = sin(phi)*sin(theta);
  SPHI  = sin(phi);
  CPHI  = cos(phi);
  STHE  = sin(theta);
  CTHE  = cos(theta);
  CAPER = cos(alpha);

  return 0;

}
```

Code 67: Configure implementation of the conic direction sampler.

Secondly, the *directionSampling* method gets an array of doubles with dimension three, which must be filled with the **normalized** sampled direction (U, V, W). Also, a random number generator is provided via the *random* parameter. The source 68 shows the implementation of the *directionSampling* function for the conic sampler.

```

void cone_directionSampling::directionSampling(double dir[3], pen_rand& random
    ) const{

  const double TWOPI = 2.0*constants::PI;

  double UT,VT,WT;
  double DF;
  double SUV;
  double UF,VF,WF;

  // Define a direction relative to the z-axis
  WT  = CAPER + (1.0-CAPER)*random.rand();
  DF  = TWOPI*random.rand();
  SUV = sqrt(1.0-WT*WT);
  UT  = SUV*cos(DF);
  VT  = SUV*sin(DF);
  // Rotate to the beam axis direction
  UF  = CPCT*UT-SPHI*VT+CPST*WT;
  VF  = SPCT*UT+CPHI*VT+SPST*WT;
  WF  =-STHE*UT+CTHE*WT;
  // Ensure normalisation
  double DXY  = UF*UF+VF*VF;
  double DXYZ = DXY+WF*WF;
  if(fabs(DXYZ-1.0) > 1.0e-14){
    double FNORM = 1.0/sqrt(DXYZ);
    dir[0] = FNORM*UF;
    dir[1] = FNORM*VF;
    dir[2] = FNORM*WF;
  }
  else{
    dir[0] = UF;
    dir[1] = VF;
    dir[2] = WF;
  }
}
```

Code 68: Conic direction sampler *directionSampling* implementation.

Notice that the *directionSampling* method is defined as constant. Therefore, this method can't change the sampler state. This is only allowed in the *configure* and *updateGeometry* calls.

7.3.1 Including files

To use a developed direction sampler in the PenRed environment, in addition to the sampler register steps described in the section 7.1, the developer must append a include of the corresponding header and source file names in the files,

`src/particleGen/direction/includes/directionSamplers.hh`

and

`src/particleGen/direction/source/directionSamplers.cpp`

respectively.

7.4 Energy

Custom `energy` samplers must inherit from the base class `abc_energySampler` as,

`public abc_energySampler`

The base class is located in the file,

`src/particleGen/includes/pen_samplers.hh`

and several examples of `energy` sampler implementations can be found in,

`src/particleGen/energy`

Energy samplers are used to obtain the initial energy (E) of the particle. As `direction` samplers, `energy` samplers have not to fill any base class member variable, but implement two functions, the *configure* and the sampling method named *energySampling*. Both function declarations are shown in the code 69.

```

virtual int configure(double& Emax, const pen_parserSection& config, const
    unsigned verbose = 0) = 0;
virtual void energySampling(double& Energy, pen_rand& random) const = 0;
```

Code 69: Energy sampler mandatory functions.

The configure method takes the `pen_parserSection` (parameter *config*) to obtain the configuration data specified by the user and the *verbose* parameter to control the print verbosity, just as the previous samplers. However, it takes also an extra parameter named *Emax*. This one is an output parameter and must be filled with the maximum energy that the sampler can return in any call of the *energySampling* method. It is important to fill this value correctly, as the maximum energy of all the sources involved in the simulation will be used to construct the required energetic grids to perform the simulations. An example of *configure* function is shown in the code 70, which corresponds to the `monoenergetic` energy sampler.

```

int monoenergetic::configure(double& Emax, const pen_parserSection& config,
    const unsigned verbose){

  int err;

  err = config.read("energy",E);
  if(err != INTDATA_SUCCESS){
    if(verbose > 0){
      printf("monoenergetic:configure:unable to read 'energy' in configuration
  . Real number expected.\n");
    }
    return -1;
  }

  if(verbose > 1)
    printf("Energy: %12.4E\n",E);

  Emax = E;

  return 0;
}
```

Code 70: Monoenergetic sampler configuration function.

Then, the *energySampling* method gets two parameters. The first one, named *Energy*, must be filled by with the sampled energy in eV. In addition, takes a instance of the `pen_random` class (parameter *random*) to be able to randomize the energy sampling. The code 71 shows the implementation of the *energySampling* method for the `monoenergetic` sampler. As it is a mono-energetic source, the random generator is not needed.

```

void monoenergetic::energySampling(double& energy, pen_rand& /*random*/) const
    {
  // Set energy
  energy = E;
}
```

Code 71: Monoenergetic sampler *energySampling* implementation.

Notice that the *energySampling* method is defined as constant. Therefore, this method can't change the sampler state. This is only allowed in the *configure* and *updateGeometry* calls.

7.4.1 Including files

To use a developed energy sampler in the PenRed environment, in addition to the sampler register steps described in the section 7.1, the developer must append a include of the corresponding header and source file names in the files,

`src/particleGen/energy/includes/energySamplers.hh`

and

`src/particleGen/energy/source/energySamplers.cpp`

respectively.

7.5 Time

Custom `time` samplers must inherit from the base class `abc_timeSampler` as,

```
public abc_timeSampler
```

The base class is located in the file,

```
src/particleGen/includes/pen_samplers.hh
```

and several examples of `time` sampler implementations can be found in,

```
src/particleGen/time
```

Time samplers are optional in the simulation, and are used to sample a initial particle time in seconds. If no time sampler is specified, the particles time will be set to 0 during the state sample. Also, notice that using a time sampler does not enables the particles time of flight calculations. This one must be enabled explicitly in the configuration file. For example, in the PenRed main program, this must be specified in the configuration file as,

```
sources/generic/source1/record-time true
```

where *source1* is the source name and can change depending on the simulation. Regarding the implementation, time samplers have not any base class member variables to fill, remaining only two required functions to implement. As other cases, these functions are the *configure* and the sampling method, named *timeSampling*. Both declarations are shown in the code 72.

```

virtual int configure(const pen_parserSection& config, const unsigned verbose = 0) = 0;
virtual void timeSampling(double& time, pen_rand& random) const = 0;
```

Code 72: Time sampler mandatory functions.

One more time, the *configure* method takes a `pen_parserSection` as first argument, which includes all the data specified by the user to configure the sampler. In addition, the parameter *verbose* must be used to control the number of prints. An example is shown in the code 73, which corresponds to the decay time sampler.

```

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

  int err;
  //Store activity and half life
  double halfLife;
  err = config.read("halfLife",halfLife);
  if(err != INTDATA_SUCCESS){
    if(verbose > 0){
      printf("decayTime:configure:unable to read 'halfLife' in configuration. Double expected\n");
    }
    return -1;
  }

  if(halfLife <= 0.0)
```

```
    return -2;

  //Get time window
  double time0;
  double time1;
  err = config.read("time/time0",time0);
  if(err != INTDATA_SUCCESS){
    if(verbose > 0){
      printf("decayTime:configure:unable to read 'time/time0' in configuration
    . Double expected\n");
    }
    return -3;
  }

  err = config.read("time/time1",time1);
  if(err != INTDATA_SUCCESS){
    if(verbose > 0){
      printf("decayTime:configure:unable to read 'time/time1' in configuration
    . Double expected\n");
    }
    return -3;
  }

  if(time0 < 0.0 || time1 < 0.0 || time1 < time0){
    if(verbose > 0){
      printf("decayTime:configure: Negative time intervals are not allowed.\n"
    );
    }
    return -4;
  }

  tau = halfLife/LOG2;

  //Calculate range of randoms
  rand0 = 1.0-exp(-time0/tau);
  drand = 1.0-exp(-time1/tau);

  if(drand >= 1.0)
    drand = 1.0-1.0e-16;

  drand -= rand0;

  if(verbose > 1){
    printf("T1/2 (s): %12.4E\n",halfLife);
    printf("tau (s) : %12.4E\n",tau);
  }

  return 0;
}
```

Code 73: Decay time sampler configuration function.

Regarding the sampling, the first parameter of the *timeSampling* method, named *time*, must be filled with the sampled time, in seconds. In addition, an instance of the class `pen_rand` is provided (parameter *random*) to be able to sample the time randomly. An example is shown in the code 74.

```

void decay_timeSampling::timeSampling(double& time, pen_rand& random) const{

  double rand = rand0+random.rand()*drand;
  time = -tau*log(1.0-rand);
}
```

Code 74: Decay time sampler *timeSampling* function.

Notice that the *timeSampling* method is defined as constant. Therefore, this method can't change the sampler state. This is only allowed in the *configure* and *updateGeometry* calls.

7.5.1 Including files

To use a developed time sampler in the PenRed environment, in addition to the sampler register steps described in the section 7.1, the developer must append a include of the corresponding header and source file names in the files,

`src/particleGen/time/includes/timeSamplers.hh`

and

`src/particleGen/time/source/timeSamplers.cpp`

respectively.

7.6 Specific

Custom `specific` samplers derive from the base class `abc_specificSampler`, which is a template class that takes a particle state as unique argument. Therefore, derived samplers must inherit as,

`public abc_specificSampler<particleState>`

where the *particleState* specify the particle state class. The base class is located in the file,

`src/particleGen/includes/pen_samplers.hh`

and several examples of `specific` sampler implementations can be found in,

`src/particleGen/specific`

Specific samplers are not like previous ones, but are a more versatile and complex kind of samplers. The first difference is, as we have seen, that specific samplers can only handle a **specific particle state**, specified by the template parameter *particleState*. However, derived types of *particleState* can also be used by the sampler. Therefore, if the generic particle state type (`pen_particleState`) is specified as the template argument, the sampler will be compatible with any particle type. This approach has been used to implement the phase space file sampler, which class declaration is shown in the code 75.

```

class psf_specificSampler : public abc_specificSampler<pen_particleState>{
  DECLARE_SAMPLER(psf_specificSampler)
  .
  .
  .
}
```

Code 75: Phase space file sampler class declaration.

7.6.1 Sampler function

Another key difference is the sampler function, named *sample*, which declaration is shown in the code 76.

```
virtual void sample(particleState& state,
          pen_KPAR& genKpar,
          unsigned long long& dhist,
          pen_rand& random) = 0;
```

Code 76: Specific sampler *sample* function.

Notice that the *sample* function is not declared as a constant method. Therefore, `specific` samplers are allowed to change their state during sampling calls. However, in return, the developer must ensure that the *sample* function is thread save to be able to use the multi-threading capabilities. To facilitate this procedure, each simulation thread uses its own specific sampler private instance, which is not shared among other threads. In addition, just in case a shared resource must be used by several threads, the base class `abc_specificSampler` provides a method to obtain a thread number identifier. This one is named *getThread* and is shown in the code 77.

```
inline unsigned getThread() const {return nthread;}
```

Code 77: Specific sampler *getThread* method.

So, the sample function is not as simple as in the previous cases. As is shown in the code 76, the first argument corresponds to the whole particle state to be filled. The second argument of type `pen_KPAR` should be filled with the sampled particle type, thus a specific sampler can produce different particle types. Particle type indexes can be found in the PenRed user manual. Then, the parameter *dhist* must be filled with the history increment. This one is supposed 1 by generic samplers, meaning that each sampled particle corresponds to a new history. However, in specific samplers this can be changed, sampling several particles in the same history or skipping more than a single history between consecutive *sample* calls. Finally, the *random* parameter provides a instance of the `pen_rand` class to be able to generate random numbers.

7.6.2 Sampling generic state

Although the sampler can change the whole state, it can delegate, partially or completely, the sampling of the generic state part to generic samplers, i.e., the position to spatial samplers, the direction to direction samplers, the energy to energy samplers and the time to time samplers.

If the developer want to delegate the sampling of some parts of the generic state to spatial, direction, energy and time samplers, this must be indicated in the constructor via the specific sampler base class constructor (code 78). This is done using a set of predefined flags, which definition is included in the code 79. The procedure will be exemplified later.

```
abc_specificSampler(__usedSamp usedSamplingsIn) :
                usedSamplings(usedSamplingsIn),
                pSpatial(nullptr),
                pDirection(nullptr),
                pEnergy(nullptr),
                pTime(nullptr)
{}
```

Code 78: Specific sampler base class constructor.

```

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

Code 79: Generic sampling usage flags for specific samplers.

Notice that the flags can be combined using the | operand, like
`(USE_SPATIAL | USE_DIRECTION)`

The flag descriptions follows,

- **USE_SPATIAL**: A generic `spatial` sampler must be used.
- **USE_DIRECTION**: A generic `direction` sampler is required.
- **USE_ENERGY**: Makes the generic `energy` sampler mandatory.
- **USE_TIME**: Requires the usage of a generic `time` sampler.
- **USE_GENERIC**: Indicates that a `spatial`, a `direction` and a `energy` generic samplers are required, but the `time` sampler is optional. Furthermore, with this flag, the generic part of the particle state will be sampled automatically as if no specific sampler was defined. Then, once the generic part has been sampled, the *sample* method of the specific sampler will be called, which can overwrite any part of the previous sampling.
- **USE_NONE**: Indicates that no generic sampler is required for this specific sampler.

Notice that if the **USE_GENERIC** flag is not set, the usage of the specified generic samplers must be handled by the specific sampler, inside the *sample* method. However, the configuration will be handled automatically. Therefore, using the flags **USE_SPATIAL**, **USE_DIRECTION** and **USE_ENERGY** does not provide the same result as using the flag **USE_GENERIC**. In the last case, the generic samplers will be handled automatically by the framework, and the specific sampler function will receive the generic state part of the *state* parameter already sampled.

Also, take into account that the flags are used to specify which generic samplers are mandatory. Therefore, if some mandatory generic sampler has not provided by the user in the configuration file, the configuration will return an error and the execution will stop. However, if a non mandatory sampler is provided by the user, it could be still used by the specific sampler, depending on its implementation. Nevertheless, a warning will be printed to inform that the corresponding generic sampler could be ignored.

Following we will discuss two examples of specific samplers. The first one, corresponding to the code 80, shows the specific sampler for polarised photons. This one delegates all the generic sampling step, indicated via the **USE_GENERIC** flag.

```

class gammaPolarised_specificSampler : public abc_specificSampler<
    pen_state_gPol>{
  DECLARE_SAMPLER(gammaPolarised_specificSampler)
  private:

```

```
  double SP10,SP20,SP30;
  int IPOL0;

  public:

  gammaPolarised_specificSampler() : abc_specificSampler<pen_state_gPol>(
    USE_GENERIC),
              SP10(0.0),
              SP20(0.0),
              SP30(0.0),
              IPOL0(0)
  {}
```

Code 80: Gamma polarised photons specific sampler class header.

Thus, its *sample* function does not require to handle the generic state part. As is shown in the code 81, the *sample* method only sets the polarisation related variables, which are specific of the state `pen_state_gPol` class.

```

void gammaPolarised_specificSampler::sample(pen_state_gPol& state,
                pen_KPAR& /*genKpar*/,
                unsigned long long& /*dhist*/,
                pen_rand& /*random*/){

  state.IPOL = IPOL0;
  state.SP1  = SP10;
  state.SP2  = SP20;
  state.SP3  = SP30;
}
```

Code 81: Gamma polarised photons *sample* function.

The next example handles completely the sample of the generic state part, and corresponds to the phase space file specific sampler. Thus, no generic sampler is required, and this is specified using the **USE_NONE** flag (code 82).

```

  psf_specificSampler() :
        abc_specificSampler<pen_particleState>(USE_NONE),
        pSF(nullptr),
        nChunks(0),
        chunksPerPart(0),
        offsetChunks(0),
        remainingChunks(0),
        buffer(nullptr),
        bufferSize(0),
        NSPLIT(1),
        WGHTL(0.0),
        WGHTU(1.0),
        RWGHTL(1.0e35),
        splitted(0),
        requiredSplits(0),
        lastKpar(ALWAYS_AT_END)
```

Code 82: Phase space file constructor.

Finally, to access the assigned `spatial`, `direction`, `energy` or `time` samplers, the `abc_specificSampler` provides a set of methods to read the pointers to these samplers, which are described in the section 7.6.6.

7.6.3 Configuration

The `specific` sampler configuration virtual method is defined in the code 83.

```

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

```
Code 83: Specific sampler configuration method.

As previous samplers, this configuration method takes a `pen_parserSection` instace, named *config*, which provides all the configuration parameters specified by the user. Also, the *verbose* parameter determines the verbosity level, i.e. the amount of printed information. As `energy` samplers, the *Emax* parameter must be filled with the maximum energy achievable by the sampler. Notice that this parameter is not necessary to be filled if a complete generic sampler is used. To enable this feature, as has been described in the section 7.6.2, the **USE_GENERIC** flag must be used in the constructor.

Notice that, when the configuration function is called, all the possible generic samplers (*spatial*, *direction*, *energy* and *time*) are already saved in the corresponding pointers. Therefore, the pointers to these samplers can be accessed, during the configuration call, using the functions described in the section 7.6.6. However, as has been discussed, the developer must take into account that the geometry can't be accessed during the *configure* method call. Therefore, no geometry related calculations should be done in the *configure* function, as the *geo* function will return a null pointer. Instead, this calculus must be done in the *updateGeometry* method (code 56), which is called when a geometry is assigned to the samplers.

7.6.4 Resume a simulation

A counterpart of allowing to modify the state of the specific sampler during the sampling method, is that the sampling can depend on the number of previous *sample* calls. For example, if the information of the states to sample are read from a file. This fact can be problematic if the simulation is resumed from a dump file. Considering the same case, if the simulation is resumed, the file will be read from the beginning, reading the same information twice.

To handle this situation, the specific sampler base class provides a virtual method named *skip*, which declaration is shown in the code 84.

```

virtual void skip(const unsigned long long /*dhists*/){}
```
Code 84: *Skip* virtual method of the `abc_specificSampler` class.

The *skip* method takes as only argument the number of histories to be skipped (parameter *dhists*). Notice that this function only must be implemented if is required by the sampler, usually because the state is changed during the *sample* call. This is the case of the phase space file sampler (class `psf_specificSampler`), as it reads the particle information from a file. Instead, the gamma polarised sampler does not implement this function, as its sampling method does not change the sampler state.

7.6.5 Dynamic generic sampling

The specific sampler base class `abc_specificSampler` provides a virtual method named *updateSamplers* (code 85). Actually, this method has not usage, but has been included to allow, in future implementations, to change the generic samplers during the simulation. Therefore, actually, this method **must not** be implemented.

```

virtual void updateSamplers(){}
```
Code 85: Virtual method *updateSamplers* of the `abc_specificSampler` class. This method must not be implemented.

7.6.6 Auxiliary functions

The base class `abc_specificSampler` provides some auxiliary methods to be able to access information to perform the sampling process. The definition of these methods are listed in the code 86

```

  inline const abc_spatialSampler* spatial() const {return pSpatial;}
  inline const abc_directionSampler* direction() const {return pDirection;}
  inline const abc_energySampler* energy() const {return pEnergy;}
  inline const abc_timeSampler* time() const {return pTime;}
  inline const wrapper_geometry* geo() const {return geometry;}
  inline unsigned getThread() const {return nthread;}
```
Code 86: Auxiliary methods provided in the `abc_specificSampler` class.

The description of each one follows,

- **spatial**: Returns a pointer to the assigned spatial `sampler`. If it has not been assigned a null pointer is returned instead.
- **direction**: Returns a pointer to the assigned `direction` sampler. If it has not been assigned a null pointer is returned instead.
- **energy**: Returns a pointer to the assigned `energy` sampler. If it has not been assigned a null pointer is returned instead.
- **time**: Returns a pointer to the assigned `time` sampler. If it has not been assigned a null pointer is returned instead.
- **geo**: Returns a pointer to the assigned geometry. If no geometry has been assigned a null pointer is returned instead.
- **getThread**: Returns a numeric identifier for the thread which owns the specific sampler instance.

Notice that to perform some generic sampling with the provided pointers, all returned samplers have a public method named *sample* which must be called instead of the overwritten method to implement the generic sampler. For example, to use a `spatial` sampler, the developer must call the *sample* method instead of the *geoSampling* method.

7.6.7 Including files

To use a developed specific sampler in the PenRed environment, in addition to the sampler register steps described in the section 7.1, the developer must append a include of the corresponding header and source file names in the files,

`src/particleGen/specific/includes/specificSamplers.hh`

and

```
src/particleGen/specific/source/specificSamplers.cpp
```

respectively.

8 Variance reduction (VR)

Variance reduction techniques (VR) are used to improve the variance reduction of the magnitudes estimated by the simulation. VR modules inherits from the class,

pen_genericVR

which source can be found in,

```
src/VR/includes/
src/VR/source/
```

This class is a template interface which takes as template argument a particle state type. Like specific sources, a VR module can be compatible with all particle states if the template argument is set to the basic particle state, i.e. `pen_particleState`. In this case, the developed VR class must inherit from the basic VR class as,

```
public pen_genericVR<pen_particleState>
```

Otherwise, a different particle state can be specified to limit the VR module usability. For example, the x-ray splitting module (code 87) is limited to photons with the state `pen_state_gPol`.

```

class pen_VRxraysplitting : public pen_genericVR<pen_state_gPol>
```

Code 87: *pen_VRxraysplitting* class declaration.

Notice that the implementation of VR modules follows the same pattern regardless the used particle state. The only difference is where the source files are registered. Therefore, the same explanation applies for both types of VR modules, generic and specific.

8.1 Register

Implemented VR modules must be registered using the *DECLARE_VR* and *REGISTER_VR* macros (code 88). These macros, takes as arguments the class name of our implemented module (*Class* parameter), the particle state type required by the module (*stateType* parameter) and a unique VR module name (*ID* parameter), which will be used in the configuration file to specify its type.

```

  DECLARE_VR(Class)
  REGISTER_VR(Class,stateType, ID)
```

Code 88: VR register macros.

The first macro must be placed inside the class declaration, just after the class name, as is shown in the code 89, which corresponds to the *pen_VRRussianRoulette* class definition.

```

class pen_VRRussianRoulette : public pen_genericVR<pen_particleState>{
  DECLARE_VR(pen_VRRussianRoulette)
.
.
.
}
```

Code 89: VR declare macro.

The second one must be placed in the source file, as is shown in the code 90.

```

REGISTER_VR(pen_VRRussianRoulette, pen_particleState, RUSSIAN_ROULETTE)
```

Code 90: VR register macro.

The source file inclusion depends on the specified particle state. Those who use the generic particle state (`pen_particleState`) are classified as generic VR modules. Instead, if another particle state is used, the module will be considered as specific.

8.1.1 Generic modules files inclusion

To use a developed generic VR module in the PenRed environment, in addition to the VR register steps described in the section 8.1, the developer must append a include of the corresponding header and source files names in the files,

`src/VR/generic/includes/genericVR.hh`

and

`src/VR/generic/source/genericVR.cpp`

respectively.

8.1.2 Specific modules files inclusion

To use a developed specific VR module in the PenRed environment, in addition to the VR register steps described in the section 8.1, the developer must append a include of the corresponding header and source files names in the files,

`src/VR/specific/includes/specificVR.hh`

and

`src/VR/specific/source/specificVR.cpp`

respectively.

8.2 VR callbacks

Like tallies (section 6), VR modules runs in some specific points during the simulation execution. These points can be enabled/disabled for each implemented VR module depending on its needs. Each of these points are handled by a specific callback, which will be described in this section. Regarding the parameters, all callbacks gets the variables shown in the code 91.

```

    const unsigned long long nhist,
    const pen_KPAR kpar,
    stateType& state,
    std::array<stateType,constants::NMS>& stack,
    unsigned& created,
    const unsigned available,
    pen_rand& random
```

Code 91: Common VR callbacks parameters.

First, *nhist* specify the history number. Secondly, *kpar* specify the particle type to which the VR technique will be applied. Then, *state* stores the particle state. Notice that, usually, this state will change during the call. Therefore, is not declared as a constant parameter. The state type is specified by the template argument (`stateType`) of the `pen_genericVR` class. As many VR techniques clones particles, like techniques based on particle splitting, the *stack* parameter provides an storage to save generated particle states. The *stack* size is limited to *constants::NMS* states. However, as the generated particles in the VR module will be included in the secondary particle stack, which is not accessible, the developer must take into account the *created* and *available* parameters. The first one (*created*) saves the number of states stored in the provided *stack* array, which could be filled previously by another VR callback. Therefore, when a new state is stored in the *stack*, the value of *created* must be increased, and the next position to save a new state is determined by the value of *created*,

```
stack[created] = newState;
++created;
```

or, more compact,

```
stack[created++] = newState;
```

Otherwise a previous created state could be overwritten. Finally, the parameter *available* stores how many free spaces are available in the secondary particle stack. This one is constant, as the secondary particle state does not change until all VR callbacks have been called. So, the remaining space can be calculated as,

$$freeSpace = available - created \tag{4}$$

Notice that the *created* parameter could store a non zero value when the callback is called, because that parameter is passed to the next VR module callback until all of them have been called. Therefore, when all callbacks have been executed, the *stack* parameter will store all the particles generated by all enabled VR callbacks, and the *created* parameter will store the total number of generated states.

Finally, the *random* parameter is a `pen_rand` instance which provides a random number generator.

All the available callbacks are listed following,

- **vr_matChange**: This function is called when the material where the particle is located changes. As extra argument, the *prevMat* parameter stores the previous material where the particle was located.

```

      void vr_matChange(const unsigned long long nhist,
        const pen_KPAR kpar,
        const unsigned prevMat,
        stateType& state,
        std::array<stateType,constants::NMS>& stack,
        unsigned& created,
        const unsigned available,
        pen_rand& random) const


```

Code 92: VR material changed callback.

- **vr_interfCross**: This function is called when the particle crosses an interface. As extra argument, the *kdet* parameter stores the detector index where the particle is located.

```

    void vr_interfCross(const unsigned long long nhist,
          const pen_KPAR kpar,
          const unsigned kdet,
          stateType& state,
          std::array<stateType,constants::NMS>& stack,
          unsigned& created,
          const unsigned available,
          pen_rand& random) const


```

Code 93: VR interface crossed callback.

- **vr_particleStack**: This function is called when a particle state is extracted from the secondary stack. As extra argument, the *kdet* parameter stores the detector index where the particle is located.

```

    void vr_particleStack(const unsigned long long nhist,
        const pen_KPAR kpar,
        const unsigned kdet,
        stateType& state,
        std::array<stateType,constants::NMS>& stack,
        unsigned& created,
        const unsigned available,
        pen_rand& random) const


```

Code 94: VR secondary particle extracted from stack callback.

Notice that all callbacks are defined as constant methods, thus changing the module state is not possible during the callback call.

8.3 Callbacks to trigger

Each VR module does not require to implement all the available callbacks. Only the necessary methods should be implemented. To specify which methods will be used by the VR module, the corresponding flags must be specified in the argument of the interface class constructor (`pen_genericVR`). These flags are,

VR_USE_PARTICLESTACK

VR_USE_MATCHANGE

VR_USE_INTERFCROSS

For example, the code 95 shows the constructor of the splitting VR module, which only requires the interface cross callback.

```
  pen_VRsplitting() : pen_genericVR(VR_USE_INTERFCROSS){
      .
      .
      .
  }
```

Code 95: VR splitting module constructor.

Notice that flags can be combined using the binary *or* operand (|). For example, to enable the *particleStack* and *matchange* flags, the constructor call must be done as shown in the code 96.

```
  VR_derived_class() :
  pen_genericVR(VR_USE_PARTICLESTACK | VR_USE_MATCHANGE){
      .
      .
      .
  }
```

Code 96: VR combined flags in module constructor.

8.4 Mandatory methods

The only mandatory method to implement in any VR module is the *configure* method (code 97). This takes as arguments a `pen_parserSection` instance named *config* which contains all the necessary information specified by the user to configure the VR module, the geometry information via the parameter *geometry*, and a *verbose* parameter to handle the print verbosity.

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

Code 97: VR configure method.

References

[1] V. Giménez-Alventosa, V. Giménez Gómez, S. Oliver, Penred: An extensible and parallel monte-carlo framework for radiation transport based on penelope, Computer Physics Communications 267 (2021) 108065. doi:https://doi.org/10.1016/j.cpc.2021.108065.
URL https://www.sciencedirect.com/science/article/pii/S0010465521001776