PenRed: Implementation Manual

V. Giménez-Alventosa[1], S. Oliver[2], and V. Giménez Gómez[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]Instituto de Seguridad Industrial, Radiofísica y Medioambiental (ISIRYM), Universitat Politècnica de València, Camí de Vera s/n, 46022, València, Spain, sanolgi@upvnet.upv.es
[3]Departament de Física Teòrica and IFIC, Universitat de València-CSIC , Dr. Moliner, 50, 46100, Burjassot, València, Spain,

September 3, 2025

Abstract

The present document is an implementation guide for the penRed framework. It is focused on how to implement new components and modules, such as sources, tallies, geometries, etc. Thanks to the modular structure of PenRed, new modules will take advantage of all its inherent features, such as multi-threading, MPI support, and compatibility with existing modules and pyPenred.

1 Introduction

This document explains, 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 internal penRed structure, but rather details the necessary methods and structures that must be implemented for each component type.

1.1 Document structure

The present document is structured as follows. First, the definition of particle states in penRed is explained in Section 2. Second, Section 3 describes the internal data structure defined in the framework and its usage; this structure is used to provide the necessary information to configure each component. Once these basic components are described, Section 5 explains how to create new geometry modules. Section 6 details how to implement and include tallies to extract information from the simulation. This is followed by modules used to generate new particle sources in Section 7. Finally, Section 8 shows how to implement variance reduction (VR) modules.

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

2 Particle state

The class `pen_particleState` stores the minimal data required to specify a particle's state. The definition of this class is located in

`src/kernel/states/pen_baseState.hh`

and it includes the following variables:

- **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 one used for polarized photons located in `src/kernel/states/gammaPol_state.hh`, all particle states must derive from the provided base state `pen_particleState`. This restriction ensures compatibility with other PenRed components.

3 Internal data format

To provide a common interface for the input data of all components, regardless of the types and amount of required data, penRed uses a set of classes to store multiple variables of multiple types in a single instance. The implementation of these classes is located in

`src/kernel/parsers/internalData`

Among these classes, `pen_parserSection` is required by almost all framework components to obtain their configuration parameters. This class uses a basic *key/value* pair format to store data, where the key structure is based on internet URLs. A generic key is a string with the following structure:

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

The value can be a number, a boolean (True or False), a character, a string, or an array of numbers, booleans, or characters. If an array is specified, it can contain multiple elements of different types.

Two different approaches can be used to extract information from this structure. The first is to read the information directly from the `pen_parserSection` structure and perform value checks manually. The second is to use a reader helper which defines how the data should be read. Both approaches will be discussed in this section.

The `pen_parserSection` implementation, and its related components, are located in

src/kernel/parsers/includes/pen_parser.hh

and

src/kernel/parsers/source/pen_parser.cpp

Additionally, the reader helper (Section 3.5) implementation is located in

src/kernel/parsers/includes/pen_reader.hh

and

src/kernel/parsers/source/pen_reader.cpp

3.1 Input text format and parsing

Users can specify all the required data to configure components via a text file where each line contains a key and a value separated by white spaces, as shown in Code 1 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

Note that for strings to be parsed correctly, their values must be enclosed in double quotes, as shown in the previous example. In addition, arrays must be enclosed by [and] with their elements separated by commas. An array example extracted from the $7-aba$ quadric example is:

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

the array is used to specify the emission probabilities. Note that an array cannot contain strings, but characters are allowed. This key/pair structure is stored in instances of `pen_parserSection` class, which implements methods to read (Section 3.2) and manipulate such data (Section **??**).

To parse files following the described format, the library includes a function named *parseFile* (Code 2), located in:

`src/kernel/parsers/internalData/includes/pen_parser.hh`

However, the parsing step is usually handled by the main program and is not required for a component development.

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

Code 2: Parse file function

Code 2 shows the function definition with its parameters, whose descriptions follow:

- **filename**: Specify the name of the file to be parsed.
- **section**: Will be filled with the parsed data once parsed.
- **errorString**: If an error occurs, a descriptive message with the error information is returned through this parameter.
- **errorLine**: If a parsing error occurs, the file line number where the error was produced is returned through this parameter.

To check if the parsing was successful, the return value of the *parseFile* function should be compared to *INTDATA_SUCCESS*, which is defined in the enumeration `pen_parserErrors`. On error, a different value is returned. The same library also includes similar functions to parse in-memory data from streams (*parseStream*) and strings (*parseString*), which exhibit the same behavior but differ in the data source. The definitions of these functions are shown in Code 3.

```
int parseStream(std::istream& sIn,
    pen_parserSection& section,
    std::string& errorString,
    long unsigned& errorLine);

int parseString(const std::string& sIn,
    pen_parserSection& section,
      std::string& errorString,
      long unsigned& errorLine);

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

Code 3: Parse file function

3.2 Reading from section

To read information from a `pen_parserSection` instance, the *read* method can be used. This method is overloaded for the allowed types: characters, integers, doubles, booleans, `pen_parserData`, strings, `pen_parserArray`, and `pen_parserSection`. These function signatures are shown in Code 4.

```

  //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 4: Parse section read functions

The *index* parameter is used only for arrays to specify the element position, and the *data* parameter is a reference where the read value will be stored. All read functions return *INTDATA_SUCCESS* if the read is successful or another value otherwise. An error is returned, for example, if the element specified by the *key* parameter is not convertible to the type of the *data* parameter.

An example of the *read* function usage is shown in Code 5. Here, the key value "energy" is read, and its value is 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 5: Read example

It is possible to read a subsection stored in a `pen_parserSection` instance and store it in a new `pen_parserSection`, as suggested by the last function in Code 4. 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 whose prefix does not match the specified key will not be copied. By default, the prefix is also removed. However, the `readSubsection` function (Code 6) can be used for the same purpose while providing the option to preserve the prefix key via the *removeKey* parameter.

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

Code 6: Read subsection function

A usage example of this function can be found in the main program, where it is used to filter the subsections corresponding to different components, such as geometry, tallies, sources, etc. Code 7 shows an example corresponding to the reading of the "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 7: 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 8) 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 8: Method **ls** from `pen_parserSection` class

For example, for this structure:

```
space/min/x 1
space/max/x 2
space/bins/x 5
space/min/y 0
space/max/y 7
space/bins/y 10
energy 1e3
```

the returned vector will contain two strings: `space` and `energy`.

3.4 Examples

Several examples of the usage of the internal format classes can be found in the configuration functions of most PenRed components (tallies, geometries, sources, etc). In addition, the tests folder,

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

contains some isolated examples.

3.5 Reader helper

To automate the reading process, error checking, and documentation generation, the internal format library provides reader helper classes. Note that this functionality is optional and still under development and will be updated in future versions to further simplify its usage.

To use these helpers, the user must implement the following elements:

- A string defining the format to be read..
- A reader class to store the read data.

Both elements are described in this section.

3.5.1 Format

The read format for a specific module or class is defined by specializing the template structure `pen_format`. This structure must store the format description in the member variable *format* and be specialized for the target class, as shown in Code 9 for the class *tallyReader_DetectionSpatialDistrib*.

```
template<>
struct pen_format<tallyReader_DetectionSpatialDistrib>{
  static constexpr const char* format = R"===(
  .
  .
  .
  )===";
};
```

Code 9: Simplified format definition structure specialization example for class tallyReader_DetectionSpatialDistrib

The format is described using the same key/value pattern as the modules configurations, as this format is internally stored in a *pen_parserSection* instance. To specify a parameter to be read, it must be defined as a section with two mandatory parameters:

- **reader-description**: Must be assigned with a text describing the parameter.
- **reader-value**: Sets the default value for the parameter. It is also used to infer the parameter's expected type.

For example, to define a parameter with the path `spatial/xmin` to be read by the helper, the Code 10 can be used. In this one, as regular configurations, the character # indicates a comment.

```
template<>
struct pen_format<reader_example>{
  static constexpr const char* format = R"===(

# Example class reader configuration

reader-description "This is an example format with a single parameter named 'spatial/xmin'"

# Spatial dimensions

## X
spatial/xmin/reader-description "Minimum X value to tally, in cm"
spatial/xmin/reader-value -1.0e35
```

```

=====";
};
```

Code 10: Example of mandatory fields for parameter definition

Note that an additional parameter named `reader-description` has been defined at the configuration root. This provides a global description for the class format, in this case for *reader_example*. With this approach, the reader will know which parameters to read from the provided configuration and their types, but not which restrictions and relationships apply to each one. These are specified by optional parameters, which are described next.

Parameter Existence

First, a parameter can be flagged as mandatory or optional for configuration, with or without conditions. To specify this attribute, the subsection `reader-required` must be defined within the parameter's format configuration. This subsection takes one or two parameters depending on the chosen option:

- **type**: Specifies the type by text. This parameter is mandatory for defining the parameter's existence requirement.
- **value**: Depending on the selected type, this parameter takes on different meanings. It is used to conditionally make a parameter either mandatory or optional. It is not required for some types.

The description of each available type, along with the corresponding **value** parameter meaning, follows:

- **required**: Flags the parameter as mandatory. No **value** is necessary for this type.
- **required_if**: Flags the parameter as conditionally required. The **value** parameter is necessary for this type. The expected **value** is a path to a boolean parameter in the configuration. If that parameter is *true*, the configured parameter is mandatory. Otherwise, it is optional.
- **required_if_exist**: Flags the parameter as conditionally required. The **value** parameter is necessary for this type. The expected **value** is a path to a parameter in the configuration. If that parameter is provided, the configured parameter is mandatory. Otherwise, it is optional.
- **optional**: Flags the parameter as optional. No **value** is necessary for this type.
- **optional_if**: Flags the parameter as conditionally optional. The **value** parameter is necessary for this type. The expected **value** is a path to a boolean parameter in the configuration. If that parameter is *true*, the configured parameter is optional. Otherwise, it is mandatory.
- **optional_if_exist**: Flags the parameter as conditionally optional. The **value** parameter is necessary for this type. The expected **value** is a path to a parameter in the configuration. If that parameter is provided, the configured parameter is optional. Otherwise, it is mandatory.

If no **reader-required** subsection is provided for a parameter, it is assigned as required by default. Code 11 expands the previous example by adding some parameters and conditions. In that code, *nx* indicates the number of bins on the X axis as an optional parameter, which defaults to *1*. Additionally, the *xmin* and *xmax* parameters are only required if the parameter *nx* is provided.

These requirements will be checked during the configuration parsing and, if not fulfilled, the corresponding error along with the involved parameter will be returned.

```
template<>
struct pen_format<reader_example>{
  static constexpr const char* format = R"===(

# Example class reader configuration

reader-description "This is an example format with a single parameter named 'spatial/xmin'"

# Spatial dimensions

## X
spatial/xmin/reader-description "Minimum X value to tally, in cm"
spatial/xmin/reader-value -1.0e35
spatial/xmin/reader-required/type "required_if_exist"
spatial/xmin/reader-required/value "spatial/nx"

spatial/xmax/reader-description "Maximum X value to tally, in cm"
spatial/xmax/reader-value 1.0e35
spatial/xmax/reader-required/type "required_if_exist"
spatial/xmax/reader-required/value "spatial/nx"

spatial/nx/reader-description "Number of bins in X axis"
spatial/nx/reader-value 1
spatial/nx/reader-required/type "optional"

===";
};
```

Code 11: Example of mandatory fields for parameter definition

Value conditions

Additionally, restrictions or conditions can be assigned to a parameter's value. These **apply only to numeric values** and are defined within the subsection **reader-conditions**. Because multiple conditions can be defined for a single parameter, each condition requires a text identifier. Similar to the `reader-required` subsection, the condition **type** and **value** must be configured for each condition, with the **value** being mandatory only for some types. An example is shown in Code 13, where the parameter **nx** from the previous example has been extended with a condition named *gt0* to ensure its value is greater than 0.

```
template<>
struct pen_format<reader_example>{
  static constexpr const char* format = R"===(

spatial/nx/reader-description "Number of bins in X axis"
spatial/nx/reader-value 1
spatial/nx/reader-conditions/gt0/type "greater"
spatial/nx/reader-conditions/gt0/value 0
spatial/nx/reader-required/type "optional"

===";
```

```
};
```

Code 12: Example of mandatory fields for parameter definition

The available condition types are described next, along with the expected meaning of the **value** parameter:

- **positive**: Defines that the parameter must be positive, meaning greater than or equal to zero. The **value** is not required for this type.
- **negative**: Defines that the parameter must be negative, meaning less than zero. The **value** is not required for this type.
- **lesser_equal**: Defines that the parameter must be less than or equal to the specified **value**. The **value** can be either a number for direct comparison or a path to an existing numeric parameter in the configuration.
- **lesser**: The same behavior as **lesser_equal** but requires a strictly lesser value.
- **greater_equal**: Defines that the parameter must be greater than or equal to the specified **value**. The **value** can be either a number for direct comparison or a path to an existing numeric parameter in the configuration.
- **greater**: The same behavior as **greater_equal** but requires a strictly greater value.
- **not_equal**: Defines that the parameter must not be equal to the specified **value**. The **value** can be either a number for direct comparison or a path to an existing numeric parameter in the configuration.
- **equal**: Defines that the parameter must be equal to the specified **value**. The **value** can be either a number for direct comparison or a path to an existing numeric parameter in the configuration.

If a specified condition is not fulfilled, an error message along with the erroneous parameter will be returned.

Complete Example

The following is the complete example used above. It corresponds to the tally identifier `DETECTION_SPATIAL_DISTRIB`, which defines an energy spectrum on a 3D spatial grid to record particles impacting a specified detector.

```
template<>
struct pen_format<tallyReader_DetectionSpatialDistrib>{
  //By default, no format is defined
  static constexpr const char* format = R"===(

# Tally "DetectionSpatialDistrib" reader configuration

reader-description "Tally to register the spatial distribution of the number of particles reaching a detector"

# Spatial dimensions

## X
spatial/xmin/reader-description "Minimum X value to tally, in cm"
spatial/xmin/reader-value -1.0e35
spatial/xmin/reader-required/type "required_if_exist"
```

```
spatial/xmin/reader-required/value "spatial/nx"

spatial/xmax/reader-description "Maximum X value to tally, in cm"
spatial/xmax/reader-value 1.0e35
spatial/xmax/reader-conditions/gt/type "greater"
spatial/xmax/reader-conditions/gt/value "spatial/xmin"
spatial/xmax/reader-required/type "required_if_exist"
spatial/xmax/reader-required/value "spatial/nx"

spatial/nx/reader-description "Number of bins in X axis"
spatial/nx/reader-value 1
spatial/nx/reader-conditions/gt0/type "greater"
spatial/nx/reader-conditions/gt0/value 0
spatial/nx/reader-required/type "optional"

## Y
spatial/ymin/reader-description "Minimum Y value to tally, in cm"
spatial/ymin/reader-value -1.0e35
spatial/ymin/reader-required/type "required_if_exist"
spatial/ymin/reader-required/value "spatial/ny"

spatial/ymax/reader-description "Maximum Y value to tally, in cm"
spatial/ymax/reader-value 1.0e35
spatial/ymax/reader-conditions/gt/type "greater"
spatial/ymax/reader-conditions/gt/value "spatial/ymin"
spatial/ymax/reader-required/type "required_if_exist"
spatial/ymax/reader-required/value "spatial/ny"

spatial/ny/reader-description "Number of bins in Y axis"
spatial/ny/reader-value 1
spatial/ny/reader-conditions/gt0/type "greater"
spatial/ny/reader-conditions/gt0/value 0
spatial/ny/reader-required/type "optional"

## Z
spatial/zmin/reader-description "Minimum Z value to tally, in cm"
spatial/zmin/reader-value -1.0e35
spatial/zmin/reader-required/type "required_if_exist"
spatial/zmin/reader-required/value "spatial/nz"

spatial/zmax/reader-description "Maximum Z value to tally, in cm"
spatial/zmax/reader-value 1.0e35
spatial/zmax/reader-conditions/gt/type "greater"
spatial/zmax/reader-conditions/gt/value "spatial/zmin"
spatial/zmax/reader-required/type "required_if_exist"
spatial/zmax/reader-required/value "spatial/nz"

spatial/nz/reader-description "Number of bins in Z axis"
spatial/nz/reader-value 1
spatial/nz/reader-conditions/gt0/type "greater"
spatial/nz/reader-conditions/gt0/value 0
spatial/nz/reader-required/type "optional"

## Detector

detector/reader-description "Detector index to tally at"
detector/reader-value 1
detector/reader-conditions/gt0/type "greater"
detector/reader-conditions/gt0/value 0

## Energy

```

```
energy/nbins/reader-description "Number of energy bins to tally spectrums"
energy/nbins/reader-value 1
energy/nbins/reader-required/type "optional"

energy/emin/reader-description "Minimum energy to be tallied"
energy/emin/reader-value 0.0
energy/emin/reader-required/type "required_if_exist"
energy/emin/reader-required/value "energy/nbins"

energy/emax/reader-description "Maximum energy to be tallied"
energy/emax/reader-value 1.0e30
energy/emax/reader-required/type "required_if_exist"
energy/emax/reader-required/value "energy/nbins"
energy/emax/reader-conditions/gt/type "greater"
energy/emax/reader-conditions/gt/value "energy/emin"

## Particle to register
particle/reader-description "Particle to be registered"
particle/reader-value "gamma"

## Print options
printBins/reader-description "Enable/disable printing bin numbers in results report"
printBins/reader-value false
printBins/reader-required/type "optional"

printCoord/reader-description "Enable/disable printing bin coordinates in results report"
printCoord/reader-value true
printCoord/reader-required/type "optional"

)===";
};
```

Code 13: Format definition example for the DETECTION_SPATIAL_DISTRIB tally

Section families

In addition to regular parameters, it is also possible to define section families or arrays. These allow the use of a wildcard within the parameter key to read multiple parameters sharing the same format. This approach is used, for example, to read multiple material configurations using a single pattern. The corresponding format is partially shown in Code 15. The complete code is located in

`src/kernel/contexts/includes/pen_context.hh`

within the `pen_format<pen_contextReaderMat>` specialization.

```

materials/${subsection}/reader-description "Sections with materials definitions"
materials/${subsection}/number/reader-value 1
materials/${subsection}/number/reader-description "Material index in the geometry system. The index 0 is reserved to void regions"
materials/${subsection}/number/reader-conditions/noVoid/type "greater"
materials/${subsection}/number/reader-conditions/noVoid/value 0

materials/${subsection}/filename/reader-description "Material filename to read/write the material information"
materials/${subsection}/filename/reader-value "-"
```

```
materials/${subsection}/filename/reader-required/type "optional_if_exist"
materials/${subsection}/filename/reader-required/value "elements"

materials/${subsection}/range/${subsection}/reader-description "Range, in cm, for the specified particle in this material. Can be disabled setting a negative value"
materials/${subsection}/range/${subsection}/reader-value -1.0
materials/${subsection}/range/${subsection}/reader-required/type "optional"

materials/${subsection}/eabs/${subsection}/reader-description "Absortpion energy, in eV, for the specified particle in this material.\nIf a negative value is set, it will be replaced by the default material value or specified range."
materials/${subsection}/eabs/${subsection}/reader-value -1.0e3
materials/${subsection}/eabs/${subsection}/reader-required/type "optional"

materials/${subsection}/C1/reader-description "C1 parameter for class II transport"
materials/${subsection}/C1/reader-value 0.05
materials/${subsection}/C1/reader-required/type "optional"
materials/${subsection}/C1/reader-conditions/positive/type "positive"

materials/${subsection}/C2/reader-description "C2 parameter for class II transport"
materials/${subsection}/C2/reader-value 0.05
materials/${subsection}/C2/reader-required/type "optional"
materials/${subsection}/C2/reader-conditions/positive/type "positive"

materials/${subsection}/WCC/reader-description "WCC parameter for class II transport. If a negative value is provided,\n WCC will be set automatically by the program."
materials/${subsection}/WCC/reader-value -1.0
materials/${subsection}/WCC/reader-required/type "optional"

materials/${subsection}/WCR/reader-description "WCR parameter for class II transport. If a negative value is provided,\n WCR will be set automatically by the program."
materials/${subsection}/WCR/reader-value -1.0
materials/${subsection}/WCR/reader-required/type "optional"
```

Code 14: Material format for parameter definition using the **subsection** wildcard

As shown, a section family is defined using the **${subsection}** wildcard within the parameter key. The reader will then expect one or more keys where the **${subsection}** wildcard is replaced by different names (e.g., a material name in the example above). Note that multiple subsection wildcards can be nested within a single key, as demonstrated for defining absorption energy and range for specific particle types within each material (Code 15).

```

materials/${subsection}/range/${subsection}/reader-description "Range, in cm, for the specified particle in this material. Can be disabled setting a negative value"
materials/${subsection}/range/${subsection}/reader-value -1.0
materials/${subsection}/range/${subsection}/reader-required/type "optional"

materials/${subsection}/eabs/${subsection}/reader-description "Absortpion energy, in eV, for the specified particle in this material.\nIf a negative value is set, it will be replaced by the default material value or specified range."
materials/${subsection}/eabs/${subsection}/reader-value -1.0e3
```

```
materials/${subsection}/eabs/${subsection}/reader-required/type "optional"
```
Code 15: Material format for absorption energy and range definition

Within a section family, the same patterns for defining parameters are used. The primary consideration involves the paths assigned to existence and condition **values**. Path resolution follows these rules:

- **Relative Paths** (without a leading slash '/'): Interpreted as relative to the current position of the **${subsection}** wildcard.

- **Absolute Paths** (with a leading slash '/'): Interpreted as relative to the global section root.

This means that within a section family, you can only reference other global parameters or those defined within the same section instance. Code 16 shows an example where the `DB-material` parameter is only required if the `DB` parameter exists within the same subsection (i.e., the same material). Similarly, the `density` is required only if the `elements` subsection exists for that material. Note that the existence check can be applied to a key path, not just a final value.

```

materials/${subsection}/DB/reader-description "Compositions database used to create the material"
materials/${subsection}/DB/reader-value "-"
materials/${subsection}/DB/reader-required/type "optional"

materials/${subsection}/DB-material/reader-description "Material name inside the composition database"
materials/${subsection}/DB-material/reader-value "-"
materials/${subsection}/DB-material/reader-required/type "required_if_exist"
materials/${subsection}/DB-material/reader-required/value "DB"

materials/${subsection}/elements/${subsection}/reader-description "The element ${subsection} must be the element atomic number (Z), and the corresponding value its fraction by weight in the created material"
materials/${subsection}/elements/${subsection}/reader-value 0.1
materials/${subsection}/elements/${subsection}/reader-required/type "optional"
materials/${subsection}/elements/${subsection}/reader-conditions/greater/type "greater"
materials/${subsection}/elements/${subsection}/reader-conditions/greater/value 0.0

materials/${subsection}/density/reader-description "Material density in g/cm**3. Only required if the material is defined in the configuration"
materials/${subsection}/density/reader-value 1.0
materials/${subsection}/density/reader-required/type "required_if_exist"
materials/${subsection}/density/reader-required/value "elements"
materials/${subsection}/density/reader-conditions/positive/type "greater"
materials/${subsection}/density/reader-conditions/positive/value 0.0
```
Code 16: Material format for composition or DB definition

3.5.2 Reader class

Once the format is defined, a reader class is needed to store the read configuration. This class must be implemented as a subclass of the `pen_configReader` template class, which takes the same subclass type as a template argument (Code 19) and inherits from `pen_readerStorage`. The latter defines the interface to be implemented by the reader class (Code 17).

```

class pen_readerStorage{

public:

  friend class pen_readerSection;

  enum errors{
    SUCCESS = 0,
    UNHANDLED = 1,
  };

  int errorCode;

protected:
  virtual inline void beginRead(){  }
  virtual inline void endRead(){  }

  virtual inline
  int beginSectionFamily(const std::string& /*pathInSection*/,
                         const size_t /*size*/,
                         const unsigned /*verbose*/)
                        { return UNHANDLED; }

  virtual inline
  int endSectionFamily(const unsigned /*verbose*/)
                      { return UNHANDLED; };

  virtual inline
  int beginSection(const std::string& /*name*/,
                   const unsigned /*verbose*/)
                   { return UNHANDLED; };

  virtual inline
  int endSection(const unsigned /*verbose*/)
               { return UNHANDLED; }

  virtual inline
  int beginArray(const std::string& /*pathInSection*/,
               const size_t /*size*/,
               const unsigned /*verbose*/)
               { return UNHANDLED; }

  virtual inline
  int endArray(const unsigned /*verbose*/)
               { return UNHANDLED; }

  virtual inline
  int storeElement(const std::string& /*pathInSection*/,
                   const pen_parserData& /*element*/,
                   const unsigned /*verbose*/)
                   { return UNHANDLED; }

  virtual inline
  int storeArrayElement(const std::string& /*pathInSection*/,
                        const pen_parserData& /*element*/,
                        const size_t /*pos*/,
               const unsigned /*verbose*/)
                       { return UNHANDLED; }

  virtual inline
  int storeString(const std::string& /*pathInSection*/,
```

```
                    const std::string& /*element*/,
                    const unsigned /*verbose*/)
                    { return UNHANDLED; }

};
```

Code 17: Definition of pen_configStorage class

These virtual methods can be implemented in a specific reader and are called automatically during the configuration read process. Each method is called in response to a specific event, allowing the reader to save the provided configuration data. Note that the read is performed in alphabetic order. Therefore, once a section read starts, all its inner parameters are read before moving to the next section at the same level. The verbose level parameter controls the amount of information provided during the read. By convention, use these values:

- **0** : Output disabled.
- **1** : Only report errors.
- **2** : Report errors, warnings, and useful data.
- **3 or greater** : Very verbose output, including debugging information.

The methods, along with their parameters, are described next:

- **beginRead**: Called when the read begins. Used for initialization purposes.
- **endRead**: Called when the read finishes.
- **beginSectionFamily**: Called when a section family (a **${subsection}** wildcard) begins. The parameters are:
  - **pathInSection**: The path up to the **${subsection}** wildcard. This is the relative path beginning after the last **${subsection}** wildcard (or the root if no previous section family exists). The specific name of the section family instance is not included.
  - **size**: The number of elements (instances) in the section family. For example, the number of material names when using the material configuration format (Code 15).
- **endSectionFamily**: Flags the end of the last started section family.
- **beginSection**: Called when a new instance of the last started section family begins. Following the example in Code 15, this method is called at the start of the configuration for each material. The instance name is provided by the **name** parameter.
- **endSection**: Flags the end of the last started section family instance.
- **beginArray**: Called when the read of a parameter storing an array begins. The parameters are:
  - **pathInSection**: The relative path to the array parameter within the last section family, if defined. From the root otherwise.
  - **size**: The number of elements in the array.

- **storeArrayElement**: Called sequentially for each array element after the **beginArray** call. The parameters are:
  - **pathInSection**: The same path as the corresponding **beginArray**.
  - **element**: The value of this array element.
  - **pos**: The index position within the array.
- **endArray**: Called when the array read finishes.
- **storeElement**: Called when a parameter value (which is neither an array nor a string) is read. The provided parameters are:
  - **pathInSection**: The relative path to the array parameter within the last section family, if defined. From the root otherwise.
  - **element**: The read value for this parameter.
- **storeString**: Called when a string parameter is read. The parameters are:
  - **pathInSection**: The relative path to the array parameter within the last section family, if defined. From the root otherwise.
  - **element**: The read string for this parameter.

Note that implementing these methods is not mandatory; they return **UNHANDLED** (1) by default. Therefore, only the methods required for a specific format need to be implemented. Each method must return an integer error code: 0 for success, 1 for unhandled (which is not an error), and any other value will be interpreted as an error, halting the configuration read and reporting the issue.

Regarding element value types, the `pen_parserData` object provides overloads to convert the value automatically. Code 18 shows the `storeElement` implementation for the `DETECTION_SPATIAL_DISTRIB` tally reader helper, which uses this feature.

```

int tallyReader_DetectionSpatialDistrib::storeElement(const std::string&
    pathInSection,
                    const pen_parserData& element,
                    const unsigned){

  if(pathInSection.compare("spatial/xmin") == 0){
    xmin = element;
  }
  else if(pathInSection.compare("spatial/xmax") == 0){
    xmax = element;
  }
  else if(pathInSection.compare("spatial/nx") == 0){
    nx = element;
  }
  else if(pathInSection.compare("spatial/ymin") == 0){
    ymin = element;
  }
  else if(pathInSection.compare("spatial/ymax") == 0){
    ymax = element;
  }
  else if(pathInSection.compare("spatial/ny") == 0){
    ny = element;
  }
  else if(pathInSection.compare("spatial/zmin") == 0){
    zmin = element;
```

```
  }
  else if(pathInSection.compare("spatial/zmax") == 0){
    zmax = element;
  }
  else if(pathInSection.compare("spatial/nz") == 0){
    nz = element;
  }
  else if(pathInSection.compare("detector") == 0){
    kdet = element;
  }
  else if(pathInSection.compare("energy/nbins") == 0){
    nEBins = element;
  }
  else if(pathInSection.compare("energy/emin") == 0){
    emin = element;
  }
  else if(pathInSection.compare("energy/emax") == 0){
    emax = element;
  }
  else if(pathInSection.compare("printBins") == 0){
    printBins = element;
  }
  else if(pathInSection.compare("printCoord") == 0){
    printCoord = element;
  }
  else{
    return errors::UNHANDLED;
  }
  return errors::SUCCESS;

}
```

Code 18: Store element implementation for the `DETECTION_SPATIAL_DISTRIB` tally reader helper

The corresponding reader helper class declaration is shown in Code 19, which includes all the necessary variables to store the data. Note that a format must be defined for the same reader class (`tallyReader_DetectionSpatialDistrib` in this example) to enable the automatic read.

```
class tallyReader_DetectionSpatialDistrib : public pen_configReader<
    tallyReader_DetectionSpatialDistrib>{

public:

  enum errors{
    SUCCESS = 0,
    UNHANDLED = 1,
    UNKNOWN_PARTICLE = 2,
  };

  double xmin, xmax;
  double ymin, ymax;
  double zmin, zmax;

  unsigned long nx, ny, nz;

  double emin, emax;
  unsigned long nEBins;

  unsigned kdet;

```

```
  unsigned ipar;

  bool printBins, printCoord;

  int storeElement(const std::string& pathInSection,
       const pen_parserData& element,
       const unsigned verbose);

  int storeString(const std::string& pathInSection,
      const std::string& element,
      const unsigned verbose);
};
```

Code 19: Reader helper declaration for the DETECTION_SPATIAL_DISTRIB tally

Reading configuration

With the reader helper class implemented and an associated format defined, it can be used to read the configured parameters from a `pen_parserSection` using the inherited `read` method, as shown in Code 20.

```
  //Read material information from config section
  tallyReader_DetectionSpatialDistrib reader;
  int err = reader.read(config,verbose);
  if(err != tallyReader_DetectionSpatialDistrib::SUCCESS){
    return err;
  }
```

Code 20: Example using the read method for a implemented reader helper

Depending on the verbose level, error, warning, and information messages will be printed during the read operation.

3.5.3 Examples

Examples of the reader helper, including both format and reader class definitions, can be found throughout the package. A basic example with no section families is the one used for the DETECTION_SPATIAL_DISTRIB tally, located in:

`src/tallies/generic/includes/tallyDetectionSpatialDistrib.hh`

and

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

A more complex example using section families is the one for material and variance reduction configurations. Their implementations are located in:

`src/kernel/contexts/includes/pen_context.hh`

and

`src/kernel/contexts/source/pen_context.cpp`

4 Automatic image export

PenRed provides an image exporter library to handle different image formats automatically. It is located in the folder:

`src/lib/image`

This library provides a single structure named `pen_imageExporter` which handles the export process via a user-defined function. This function must return a specific image pixel value when called and is provided as a constructor argument. All available constructors are shown in Code 21. As can be seen, one constructor is defined for each compatible return variable type. Furthermore, two different kinds of functions can be provided for the same variable type, depending on whether the function returns an associated uncertainty for the value or not. Both will be discussed next.

```
// ** 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 21: Image exporter constructors

4.1 User provided functions

First, the description of functions without associated uncertainties is provided, as their parameters are common to both function types. The arguments passed to these functions are shown in the example in Code 24, which is used to export 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 22: Image exporter constructor example for DICOM contour masks

Although the return type can vary, the arguments are the same. The first argument, of type *unsigned long long*, is intended to be the number of simulated histories (which is not used in this example). The second parameter, *i*, specifies the pixel to be rendered; in the example, it corresponds to the index within the mask vector. Note also the cast performed on the returned value. As the return type must match one of the types in the available constructors (Code 21), it has been cast from unsigned char to `uint8_t`.

Once constructed, certain values must be set in the `pen_imageExporter` instance to export the image properly. First, a base name must be provided to assign a filename to the exported image. This is done via the public variable *baseName*, which is a `std::string`. Then, the number of dimensions, the number of elements in each dimension, and the element size in each dimension must be specified using the member function *setDimensions*. Using the DICOM mask exporter example again, the number of dimensions is 3 (x, y, z), and the number of elements and their size coincide with the number and size of voxels in the DICOM, as shown in Code **??**. Additionally, an origin can 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 23: Image exporter configuration for DICOM contour masks

Regarding functions with associated uncertainties, they use the same initial arguments discussed above, plus an additional parameter. This extra parameter is a reference to store the associated uncertainty of the returned value, and its type must match the return type. An example is shown in 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 24: Image exporter constructor example for DICOM contour masks

4.2 Available formats

Currently only two image formats:

- **MHD** (MetaImage Header Data): Typically associated with medical imaging and is used to describe multi-dimensional images. It consists of two files:
  - **Header**: The *.mhd* file contains metadata that provides information about the image's dimensions, voxel size, data type, and other details.
  - **Data**: The .mhd file typically references the corresponding raw data file, which stores the pixel/voxel data. The raw data file usually has the extension *.raw*, which is the one used by the exporter.
- **GNU**: This export format creates a file to be plotted using gnuplot with the image matrix option.

5 Geometry

Geometry modules are used both to locate particles within the geometry system and to move them through it. 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` requires defining many functions whose behavior is often repeated, depending on the geometry type.

To simplify implementation, penRed provides two high-level base classes (`abc_geometry` and `abc_mesh`) for developing geometry modules, categorized by the elements that constitute the geometry system.

The first case involves geometries constructed from volumetric **objects**, where each object is defined by a single material and encloses a volume filled with that material. These objects can, in turn, contain other bodies. The method used to describe these objects is not restricted; they can be defined, for example, by quadric surfaces, a triangulated surface mesh, etc.

The second case involves geometries constructed using a volumetric mesh, where each element (e.g., a voxel) is assigned an independent index, material, and density. In this type, objects are not explicitly defined. An example of this type is a geometry built from DICOM images using a voxelized 3D mesh.

The following sections explain the procedure for implementing each type of geometry. First, however, we will discuss some common information and mandatory methods that must be defined by the developer, regardless of the geometry type.

5.1 Generic assumptions

The following assumptions must be satisfied by any new geometry module:

- A geometry module must be composed of **bodies**. Each body is filled with a single **material**, identified by an unsigned integer index. Additionally, bodies can belong to a **detector**, which is also identified by an unsigned integer index.
- The **material** index 0 is reserved for void regions.
- **Detector** assignment is optional. If a body does not belong to any detector, its detector index must be set to 0.
- An **interface** is defined as a boundary between bodies where the **material** index or the **detector** index changes. Therefore, if two adjacent bodies share the same **material** index, there is no interface between them unless they belong to different **detectors**.
- A particle must be stopped when it reaches an **interface**, unless the **material** after the interface is void. In that case, the particle must be moved through 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 a particle escapes from the geometry system, it is moved an "infinite" distance, typically on the order of 10^{35} cm.

5.2 Mandatory functions

The `abc_geometry` and `abc_mesh` classes define most of the pure virtual methods inherited from the `wrapper_geometry` class for body-based and mesh-based geometries, respectively. However, some methods must still be implemented by the geometry developer in both cases. These methods are listed below:

- **getIBody** (Code 25): This method takes a body name (*elementName*) as its only argument. It must return the body index identified by this name. If the name does not correspond to any defined body, the total number of bodies must be returned instead.

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

Code 25: Declaration of the pure virtual method *getIBody* from the `wrapper_geometry` class.

For example, in the quadric-based geometry (class `pen_quadricGeo`), the array *BALIAS* belonging to each body is used for identification. The corresponding `getIBody` implementation is shown in Code 26.

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

- **getBodyName** (Code 27): The counterpart to **getIBody** must also be provided. This function retrieves the body name corresponding to the specified input index (*ibody*).

```
virtual std::string getBodyName(const unsigned ibody) const = 0;
```

Code 27: Declaration of the pure virtual method *getBodyName* from the `wrapper_geometry` class.

- **locate** (Code 28): The *locate* method handles the localization of a particle inside the geometry system. It takes the particle state as its only parameter and must update the body (*IBODY*) and material (*MAT*) indexes of the input state accordingly. Usually, only the position (X, Y, Z) is required for this purpose, but the entire state is accessible.

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

Code 28: Declaration of the pure virtual method *locate* from the `wrapper_geometry` class.

If 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 than or equal to the number of bodies. An example is shown in Code 29, which defines the **locate** method for the voxelized geometry class `pen_voxelGeo`.

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

As shown, **locate** is defined as a constant method. Therefore, it is not possible to change the geometry state during this function call. This restriction ensures the geometry state remains constant during simulation, allowing sharing it among all threads.

- **step** (Code 30): The **step** method handles the movement of a particle inside the geometry system. To achieve this, all the assumptions from Section 5.1 must be satisfied.

```

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

Code 30: Declaration of the pure virtual method *step* from the `wrapper_geometry` class.

The **step** method takes several parameters, which are described below:

  – **state**: The particle state to be moved. During the call, the state must be updated to move the particle. The body (*IBODY*) and material (*MAT*) indexes must also be updated if they change.
  – **DS**: The input parameter storing the maximum distance to be traveled by the particle. The actual traveled distance may be less if an interface is crossed. Note that void regions must be skipped even if **DS** is less than the distance required to cross a void region.
  – **DSEF**: This output parameter must be set to the distance traveled in the original material, i.e., the material index stored in the **state** before moving the particle. If the particle is located in a non-void region and crosses an interface into a void region, the distance traveled in the void region must not be added to **DSEF**. However, if the particle is initially in a void region, **DSEF** must score the distance traveled until the next non-void region or until the particle escapes the geometry system.

- **DSTOT**: This output parameter must be set to the total traveled distance, regardless of whether a void region has been crossed. Therefore, if the particle does not reach an interface or the material after the interface is not void, the relation $DSTOT = DSEF$ must hold. Conversely, if the particle moves from a non-void region and crosses an interface into a void material, then $DSTOT > DSEF$ must be true, because **DSTOT** must include both the distance in the original material (**DSEF**) and the distance traveled in the void region.
- **NCROSS**: This output parameter counts the number of interfaces crossed. 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.

Like **locate**, the **step** method is defined as constant. Therefore, it is not possible to change the geometry state during this function call. Code 31 shows the simplest possible **step** function, where the entire space is filled by a single material and no interfaces 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 31: Declaration of the *step* method of the `pen_dummyGeo` class.

- **configure** (Code 34): The configuration function takes as its first argument a `pen_parserSection` (Section 3) instance containing all the information provided by the user to configure the geometry. Additionally, the main program includes information about the configured materials in a section named *materials*. Currently, this section stores the index and density of each configured material, but this information could be extended in future implementations. Code 32 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 32: 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 Code 33.

```cpp

//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 33: Geometry material information read in the DICOM configuration function.

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

```cpp

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

Code 34: Declaration of the pure virtual method *configure* for the `wrapper_geometry` class.

During configuration, the variable *configStatus* must be set to a value indicating the result. If the geometry is configured successfully, both *configStatus* and the return value must be 0. A non-zero value in either *configStatus* **or** the return value will be interpreted as a failed configuration.

Additionally, depending on the geometry type, specific variables must be assigned to ensure the module works properly. These variables will be explained in the specific sections corresponding to body-based (Section 5.4) and mesh-based (Section 5.5) geometries.

5.3 Registering

To use a developed geometry module in the penRed environment, regardless of its type, the macros DECLARE_GEOMETRY and REGISTER_GEOMETRY must be used (Code 35).

```

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

Code 35: Provided registration geometry macros.

The parameters for these macros are the class name (parameter *Class*) and a text identifier used to specify the geometry type in the configuration file (parameter *ID*).

The first macro, DECLARE_GEOMETRY, must be placed inside the class definition, immediately after the class name, as shown in the example in Code 36.

```

class pen_quadricGeo : public abc_geometry<pen_quadBody>{
  DECLARE_GEOMETRY(pen_quadricGeo)
   .
   .
   .
}
```

Code 36: 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 shown in Code 37.

```

REGISTER_GEOMETRY(pen_voxelGeo,VOXEL)
```

Code 37: Usage of the REGISTER_GEOMETRY in the pen_voxelGeo geometry source file.

Additionally, 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 subsections.

5.4 Body based

Body-based geometries inherit from the template abstract class abc_geometry, located in the file:

src/geometry/includes/geometry_classes.hh

Geometries derived from this class must use the inheritance:

public abc_geometry<bodyType>

where the template parameter (*bodyType*) specifies the type of the bodies used to construct the geometry. 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 fulfill the specific body geometry requirements. However, it is advisable to define the body type as a class derived from pen_baseBody (Code 38), which defines the minimum interface required for compatibility with all predefined methods of 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 38: Definition of `pen_baseBody` structure.

The required variables for any body are declared in the `pen_baseBody` class and are described below:

- **MATER:** Assigned material index.
- **KDET:** Assigned detector index.
- **DSMAX:** Maximum allowed step length for particles with soft energy deposition (electrons and positrons).
- **localEABS:** Array storing the local absorption energies for each particle type. Note that the material absorption energy is also considered to determine the final absorption energy. The most restrictive energy will be used for each body. If $matEABS$ is the material absorption energy, the final absorption energy ($EABS$) is set as:

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

An example of an extended body type can be found in the definition for the quadric geometry implementation, named `pen_quadBody` (Code 39), 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 39: Definition of `pen_quadBody` structure.

5.4.2 Inherited variables

The `abc_geometry` class provides several variables that must be used to ensure the proper operation of the developed geometry. These variables are defined as follows:

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

  `src/kernel/includes/pen_constants.hh`

- **NBODYS:** An `unsigned integer` variable that stores the number of bodies in the current geometry. It is initialized to 0 and must be updated during the geometry configuration.

- **bodies:** An array of bodies with dimension **NB**. The array type corresponds to the geometry's body type, specified by the 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 registration steps described in Section 5.3, the developer must include 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. Additionally, in the header file, the geometry class name must be added to the tuple type *typesObjectGeos* (Code 43).

```
  using typesObjectGeos = std::tuple<pen_quadricGeo,
              pen_dummyGeo,
              pen_meshBodyGeo,
              pen_comboGeo,
              pen_filterGeo>;
```

Code 40: Definition of `typesObjectGeos` type.

5.5 Mesh based

Mesh-based geometries inherit from the template abstract class `abc_mesh`, located in the file:

`src/geometry/includes/geometry_classes.hh`

Geometries derived from this class must use the inheritance:

```
public abc_mesh<meshElement>
```

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

In mesh-based geometries, to ensure compatibility with all tallies and sources, each non-void material is considered an independent body. Thus, the **IBODY** index can be obtained as:

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

for **non-void** materials. Additionally, body index 0 is reserved for an enclosure that is intended to surround the entire mesh. This enclosure is created to account for backscattering effects at the mesh boundaries. Depending on the implementation, the enclosure can be treated as a detector. This approach forces an interface between the enclosure and the mesh, regardless of the mesh element material. 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 specific geometry requirements. However, it should derive from the provided base type `pen_baseMesh`, whose definition is shown in Code 41. This base type provides the minimum interface required for compatibility with all predefined methods of the `abc_mesh` class.

```
struct pen_baseMesh{

  unsigned int MATER;

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

Code 41: Definition of `pen_quadBody` structure.

As shown, the only requirement for a mesh element is a material index (*MATER*). This is because, as discussed, mesh-based geometries identify each non-void material with a body. Thus, each mesh element is not a body itself, but a portion of a body. For this reason, the default mesh element does not have a detector index or local energy absorption, as these variables are intended to be set per body. Moreover, in meshes, defining a local energy absorption for each body would produce the same effect as defining the same energy absorption for the entire material.

Therefore, 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, discussed in the next section.

5.5.2 Inherited variables

The `abc_mesh` class provides several variables that must be used to ensure the proper operation of the developed geometry. These variables are defined as follows:

- **mesh**: A pointer whose type matches the mesh element type defined by 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 42), which takes the new mesh dimension as its only parameter. Note that the mesh is completely cleared when resized; all stored elements are destroyed.

```
void resizeMesh(unsigned dim);
```

Code 42: *resizeMesh* method `pen_quadBody` structure.

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

- **meshDim**: This variable stores the capacity of the *mesh* array, i.e., the number of elements it can store. The value of **meshDim** must not be changed manually, as it is handled automatically by the **resizeMesh** method.
- **nElements**: Must store the number of actual **mesh** elements. Therefore, it must be lesser or equal to **meshDim**. The developer must set this value during the geometry configuration.
- **DSMAX**: An array with a dimension equal to 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. Note that a body consists of all elements with the same non-void material. Therefore, the body index can be obtained from Equation 2. This array must be filled during the geometry configuration.
- **KDET**: An array with a dimension equal to the maximum number of allowed materials (*constants::MAXMAT*). Each array element stores the detector number for the corresponding body. This array must be filled during the geometry configuration.
- **nBodies**: A variable to store the total 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 registration steps described in Section 5.3, the developer must include 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. Additionally, in the header file, the geometry class name must be added to the tuple type *typesMeshGeos* (Code 43). Note that the geometry must be added to both definitions (with and without DICOM enabled) to be available under all compilation conditions.

```
#ifdef _PEN_USE_DICOM_
    using typesMeshGeos = std::tuple<pen_voxelGeo, pen_dicomGeo>;
#else
    using typesMeshGeos = std::tuple<pen_voxelGeo>;
#endif
```

Code 43: Definition of `typesMeshGeos` type.

6 Tallies

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

`pen_genericTally`

whose implementation can be found in the files:
`src/tallies/includes/pen_tallies.hh`
`src/tallies/source/pen_tallies.cpp`

Note that `pen_genericTally` is a template class that takes the particle state type as its only argument. However, currently, only tallies using 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` is used. However, existing tallies can also serve as examples. These are located in:
`src/tallies/generic/includes`
`src/tallies/generic/source`

6.1 Extracting simulation data

The functionality of tallies (and other components) is specified via callbacks defined as virtual functions. These callbacks are invoked during the simulation at specific points. To create a new tally, the developer must know when these callbacks are called and what information they receive.

Before describing the callbacks, it is useful to discuss some arguments common to most of these functions, listed in Code 44:

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

Code 44: Common variables for tally functions.

where *nhist* stores the particle history number, *kdet* the detector index where the particle is located, *kpar* the particle type index, and *state* the current particle state. Particle indexes can be found in the *Particle indexes* subsection of the *Constants, parameters and definitions* section in the usage documentation.

Not all functions receive all these parameters; some require additional specific parameters. The available callbacks and their parameters are described below. Code from the example tally `pen_tallyDummyLog` is provided for each callback.

- **tally_beginSim**: This function takes no arguments and is called when the global simulation begins (i.e., not for each new history). It is not called when a new source simulation begins. Note that if the simulation is resumed from a dump file, this function will be called at the beginning of the resumed simulation.

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

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

```

Code 45: Dummy logger `tally_beginSim` function.

- **tally_endSim**: Called when the global simulation ends. This function is expected to be called only once per simulation. It takes the number of the last simulated history as its 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 46: Dummy logger `tally_endSim` function.

- **tally_move2geo**: Called when a particle sampled by a source into a void region is moved forward to check if it reaches any non-void volume via the geometry's *step* function. This function is called *after* moving the particle. If the particle remains in a void material, it has escaped the geometry system. If it reaches a non-void volume, the simulation will continue (the particle simulation will be considered begun). Otherwise, the particle simulation never begins and is considered ended. Therefore, this function is called when the particle simulation has not started. Note that this function is only called for particles sampled in a void region; it is not triggered for particles sampled in non-void regions.

  The extra variables *dsef* and *dstot* provide the corresponding output values from the *step* function. As the particle is initially in a void region, *dsef* and *dstot* store the same value, but both are provided for future compatibility.

```
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 volume and, then, is
  //moved to find a geometry non void volume.

  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 47: Dummy logger `tally_move2geo` function.

- **tally_sampledPart**: Called when a new particle is sampled. The state of the sampled particle is exactly as created by the sampler function. If the particle is created in a void volume ($MAT = 0$), this function is called *before* attempting to move it to a non-void volume. Thus, **tally_sampledPart** is triggered before **tally_move2geo**. The parameter *dhist* indicates the number of histories skipped since the previous sampled particle. Note that a single sampler can create several particles within the same history and can also skip multiple histories in a single sample call (e.g., this behavior is common with the phase space file (PSF) sampler).

```
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 48: Dummy logger `tally_sampledPart` function.

- **tally_endHist**: Called when a history ends its simulation (i.e., the primary particle and all its generated secondary particles have 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 49: Dummy logger `tally_endHist` function.

- **tally_beginPart**: Called when a particle simulation begins. As explained for **tally_move2geo**, a particle simulation begins only if the particle reaches a non-void region of the geometry system. Therefore, if a particle is sampled in a void region and remains there after attempting to move it, 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 50: Dummy logger `tally_beginPart` function.

- **tally_endPart**: Called when a particle simulation ends or can not be started. Unlike **tally_beginPart**, this callback is triggered even for particles sampled in a void region that never reach a non-void region.

```
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 51: Dummy logger `tally_endPart` function.

- **tally_localEdep**: Called when a particle loses energy locally during its simulation (e.g., during interactions or particle absorption). It is *not* triggered for energy losses from soft interactions during a step. Note that part of the lost energy may be used to create new particles, so the lost energy is not necessarily the energy absorbed by the material. To measure absorbed energy properly, the energy of generated secondary particles must be considered in the **tally_beginPart** callback. The argument *dE* stores the amount of energy deposited (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 52: Dummy logger `tally_localEdep` function.

- **tally_step**: Called after a *step* call during particle simulation. This function is called after the particle age is updated (i.e., after *dpage* is called). It is *not* triggered when *step* is used to move a newly sampled particle from a void volume (that case uses **tally_move2geo**), because the simulation has not yet started. Therefore, **tally_step** is triggered only after the particle simulation has begun.

  An extra argument, *stepData*, is received. Its type is the structure `tally_StepData`, defined in Code 53.

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

```

Code 53: `tally_StepData` structure.

The variables *dsef* and *dstot* store the corresponding output values from the *step* function. *softDE* stores the energy deposited during the step. This energy was deposited at the point (*softX, softY, softZ*). Finally, *originIBODY* and *originMAT* save the particle's *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 54: Dummy logger `tally_step` function.

- **tally_interfCross**: Called when a particle crosses an interface during its simulation (i.e., when the *step* method returns a non-zero *NCROSS*). Moving particles sampled in void regions to the geometry does 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 55: Dummy logger `tally_interfCross` function.

- **tally_matChange**: Called when the particle changes material during simulation. Moving particles sampled in void regions to the geometry does not trigger this function. An extra argument, *prevMat*, is provided storing the previous material before crossing the interface.

```
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 56: Dummy logger `tally_matChange` function.

- **tally_jump**: Triggered immediately after each *jump* call during 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 57: Dummy logger `tally_jump` function.

- **tally_knock**: Triggered immediately after a *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 58: Dummy logger `tally_knock` function.

- **tally_lastHist**: Triggered to update the number of previous histories (e.g., on source changes or for resumed simulations). Tallies that use information about the last registered history, such as those creating phase space files, require this information to function correctly.

```
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 59: Dummy logger `tally_lastHist` function.

6.2 Callbacks to trigger

Despite the number of available tally functions, a single tally does not need to implement all of them. Only the functions necessary to collect the required data should be implemented. To specify which functions a tally will use, the corresponding flags must be passed as an argument to the `pen_genericTally` interface class 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 specifies the corresponding function. As an example, Code 60 shows the constructor of the material energy deposition tally (`pen_EdepMat` class), which includes the flags for all necessary callbacks.

```
pen_EdepMat::pen_EdepMat() : pen_genericTally( USE_LOCALEDEP    |
                                               USE_BEGINPART    |
                                               USE_SAMPLEDPART  |
                                               USE_STEP         |
                                               USE_ENDHIST      |
                                               USE_MOVE2GEO)
```

Code 60: 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. While our dummy logger doesn't extract any information and thus some of these functions do nothing in its class, several examples can be found in the built-in penRed tallies. These methods are:

- **saveData**: Called when a data report occurs. This function must store the data of interest in the corresponding data files. It receives the number of simulated histories as the only parameter. It is not necessary to manage output file names, as the tally cluster will automatically add a prefix to all created files based on the tally name, thread identifier, and MPI rank number. Note that the *saveData* function is expected to be called only after a history finishes its simulation. Its signature is shown in Code 61.

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

Code 61: Tally **saveData** function.

- **flush**: This function handles, if needed, the calculation of final results. Tallies often use auxiliary temporal data buffers to avoid recalculating unchanged bins. After the flush call, all 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, just before the *saveData* function. Its signature is shown in Code 62.

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

Code 62: Tally **flush** function.

- **sumTally**: This function receives a second tally instance of the same type to sum their results. The sum must be stored in the tally instance that calls *sumTally*, not in the argument instance. On success, this function should return 0. This function is used, for example, to aggregate results from different threads in multi-threaded simulations. Its signature is shown in Code 63. The *tallyType* must be the same as the developed tally class type.

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

Code 63: Tally **sumTally** function. The *tallyType* type must coincide with the developed tally class type.

- **configure**: Like other components, tallies require a configuration initialization function. This function takes a configuration structure, a verbose level, a geometry wrapper, and material information as arguments, as shown in Code 64.

```cpp

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

Code 64: Tally configuration function.

The *configure* function must parse and save the user-specified parameters from the `pen_parserSection` structure, described in Section 3. The *verbose* parameter specifies the output verbosity level.

The *geometry* and *materials* arguments provide all the simulation geometry and material information that could be used by the tally. For example, a tally could be specific to a certain geometry or use this information to obtain the mass of virtual mesh elements for calculating absorbed dose. Note that the material with index N in the geometry system corresponds to the position $N-1$ in the *materials* array, because the void material ($MAT = 0$) is not included in the array.

Additionally, 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* object handles writing and reading the tally state in binary format. These processes are automatically managed by the tally cluster, but the developer must specify which variables need to be dumped. To register variables for saving and loading, the developer must use the method *toDump*. An example is shown in Code 65.

```cpp

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

Code 65: Tally *dump* registration for `tallyEnergyDepositionMat`.

As shown, only one function is required to register the tally data. The *toDump* function registers the data for both writing and reading binary dumps. It takes two arguments: a pointer to the data to register, and the number of elements to register from that pointer. Note that *toDump* is overloaded to accept basic data types to simplify its usage, but it cannot handle user-defined structures.

6.4 Optional non callback methods

In addition to the mandatory methods, there are some implementable methods that provide access to specific information and features. These are optional and can be omitted if not required for a particular tally.

- **sharedConfig**: This function receives a second tally instance of the same type to retrieve values after configuration or perform any other task. It is called immediately after the configuration function. It is only called for tallies with a thread identifier greater than 0, and it receives the corresponding tally with thread identifier 0 as its argument. Therefore, this function exists to allow expensive configuration calculations to be performed once in the thread 0 tally and then shared with other threads, avoiding redundant calculations and data duplication. The function must return a value of 0 on success. Its signature is shown in Code 69.

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

Code 66: Tally **sharedConfig** function. Notice that the *tallyType* type must coincide with the developed tally class type.

An example is shown in Code 67, corresponding to the kerma track length tally. In this example, the coefficients calculated by the thread 0 are shared with the other ones.

```
int pen_tallyKermaTrackLength::sharedConfig(const
    pen_tallyKermaTrackLength& tally){

  //Set muen pointers to shared data and copy active materials
  for(unsigned imat = 0; imat <= constants::MAXMAT; ++imat){
    muen[imat] = tally.muen[imat];
    activeMat[imat] = tally.activeMat[imat];
  }

  return 0;
}
```

Code 67: Implementation of **sharedConfig** function in the pen_tallyKermaTrackLength class.

6.5 Auxiliary functions

The following helper functions are already implemented and available for use in tally development:

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

```
inline unsigned getThread() const
```

Code 68: Tally **getThread** function.

- **readStack**: Takes a *kpar* value as an argument and returns a read-only pointer to the particle stack corresponding to this *kpar*. Note that the stack state cannot be modified. Also note that stacks are only available during simulation callbacks and cannot be used within the configuration function.

```
inline const abc_particleStack* readStack(const pen_KPAR kpar) const
```

Code 69: Tally **readStack** function.

6.6 Register

Implemented tallies must be registered using the macros:
DECLARE_TALLY

and

REGISTER_COMMON_TALLY

(Code 70). These macros require three mandatory parameters: 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). The macros are called using these parameters:

```

  DECLARE_TALLY(Class,State,ID)
  REGISTER_COMMON_TALLY(Class)
```

Code 70: Tally register macros.

The first macro must be placed inside the class declaration, immediately after the class name, as shown in Code 71. This code corresponds to the `pen_tallyDummyLog` class definition.

```

class pen_tallyDummyLog : public pen_genericTally<pen_particleState> {
  DECLARE_TALLY(pen_tallyDummyLog, pen_particleState, DUMMY_LOG_EXAMPLE)
.
.
.
}
```

Code 71: Tally `pen_tallyDummyLog` declaration macro.

The second macro must be placed in the source file, as shown in Code 72.

```

REGISTER_COMMON_TALLY(pen_tallyDummyLog)
```

Code 72: Tally `pen_tallyDummyLog` register macro.

Finally, the header and source files must be included in the files:

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

and

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

respectively. Additionally, in the header file, the tally class name must be added to the tuple type *typesGenericTallies* (Code 73). Note that the tally must be added to both definitions (with and without DICOM enabled) to be available under all compilation conditions.

```
#ifdef _PEN_USE_DICOM_
    using typesGenericTallies = std::tuple<pen_EdepMat,
                   pen_EdepBody,
                   pen_SphericalDoseDistrib,
                   pen_tallyTracking,
                   pen_EmergingPartDistrib,
                   pen_CylindricalDoseDistrib,
                   pen_ImpactDetector,
                   pen_tallySecondary,
                   pen_SpatialDoseDistrib,
                   pen_AngularDet,
                   pen_tallyPhaseSpaceFile,
                   pen_tallyKermaTrackLength,
                   pen_DICOMDoseDistrib,
                   pen_CTsinogram,
                   pen_tallyDICOMkerma,
                   pen_PSS,
                   pen_DetectionSpatialDistrib,
                   pen_EmergingSphericalDistrib,
                   pen_Singles,
                   pen_tallyTrackingMemory>;
#else
    using typesGenericTallies = std::tuple<pen_EdepMat,
```

```
                  pen_EdepBody,
                  pen_SphericalDoseDistrib,
                  pen_tallyTracking,
                  pen_EmergingPartDistrib,
                  pen_CylindricalDoseDistrib,
                  pen_ImpactDetector,
                  pen_tallySecondary,
                  pen_SpatialDoseDistrib,
                  pen_AngularDet,
                  pen_tallyPhaseSpaceFile,
                  pen_tallyKermaTrackLength,
                  pen_CTsinogram,
                  pen_PSS,
                  pen_DetectionSpatialDistrib,
                  pen_EmergingSphericalDistrib,
                  pen_Singles,
                  pen_tallyTrackingMemory>;
#endif
```

Code 73: Definition of `typesGenericTallies` type.

6.7 In-memory outputs

At this point, our tally is capable of extracting information from the simulation and saving it to files on disk. However, it is also advisable to define how the output can be retrieved in memory. This allows C++ and pyPenred-based programs to extract the tally output directly from the simulation object instead of reading it from a file. The procedure to achieve this is described below.

6.7.1 Declaring outputs

First, the type and dimensions of the output must be specified via the `DECLARE_TALLY` macro by adding one parameter per output. The parameter to declare an output is a C++ standard pair with the following signature:

```
std::pair<outputType, penred::tally::Dim<N>>
```

where *outputType* specifies the value type and N specifies the number of dimensions. For example, Code 74 shows the output definition for the `DICOM_KERMA_TRACK_LENGTH` tally. This tally provides two outputs: a 3D spatial mesh with kerma values and a 2D matrix with DVH curves, where the dimensions correspond to absorbed dose and volume percentage.

```
class pen_tallyDICOMkerma: public pen_genericTally<pen_particleState> {
  DECLARE_TALLY(pen_tallyDICOMkerma,pen_particleState,DICOM_KERMA_TRACK_LENGTH
    ,
    std::pair<double, penred::tally::Dim<3>>, //Kerma
    std::pair<double, penred::tally::Dim<2>>  //DVH
    )
```

Code 74: Definition of `DICOM_KERMA_TRACK_LENGTH` outputs.

Note that no comma should be placed after the last output declaration, as this will cause a compilation error. Additionally, when using multiple outputs, the order in the macro matters, as each output is assigned an index matching this declaration order.

Depending on the value type and the number of dimensions, the output must be provided as either a standard `std::vector` or a `penred::measurements::results` type:

- If the value type is not arithmetic or the dimension is set to 0, a `std::vector<outputType>` is expected to provide the output results.

- If the value type is arithmetic and the dimension is 1 or greater, a `penred::measurements::results<ou` `N>` is expected to provide the output results.

6.7.2 Defining outputs

To create the declared outputs, the tally must provide functions to generate the corresponding structure (std::vector or penred::measurements::results) based on the requested type and dimensions. These functions are provided via the `setResultsGenerator` method, whose signature is shown in Code 75. The type resolved by the `std::tuple_element` call is a `std::function` wrapper. The wrapped function must return a std::vector¡outputType¿ or penred::measurements::results¡outputType, N¿, depending on the declared output, and takes a single argument of type `unsigned long long`, corresponding to the number of simulated histories. The template parameter I specifies the output index, according to the declaration order in the `DECLARE_TALLY` macro.

```
template<size_t I>
void setResultsGenerator(std::tuple_element_t<I, ResultsGeneratorType> f)
```

Code 75: `setResultsGenerator` function signature.

A simple approach to define these functions is to call `setResultsGenerator` in the constructor using a lambda function as argument. Code 76 shows how the results generation function is defined for output number 0, corresponding to a Cartesian 3D mesh where kerma is scored.

```
  //Register results functions
  setResultsGenerator<0>
    ([this](const unsigned long long nhists) -> penred::measurements::results<
    double, 3>{

      double invn = 1.0/static_cast<double>(nhists);
      //Create results
      penred::measurements::results<double, 3> results;
      results.initFromLists
  ({static_cast<unsigned long>(nbinsCart.x),
     static_cast<unsigned long>(nbinsCart.y),
     static_cast<unsigned long>(nbinsCart.z)},
    {penred::measurements::limitsType(minsCart.x, maxsCart.x),
     penred::measurements::limitsType(minsCart.y, maxsCart.y),
     penred::measurements::limitsType(minsCart.z, maxsCart.z)
    });

      results.description =
  "PenRed: Cartesian kerma report";

      results.setDimHeader(0, "x (cm)");
      results.setDimHeader(1, "y (cm)");
      results.setDimHeader(2, "z (cm)");
      results.setValueHeader("Kerma (eV/g hist)");

      const size_t nBins = nbinsCart.x*nbinsCart.y*nbinsCart.z;
      for(size_t i = 0; i < nBins; ++i){
  double q = cartesian[i]*invn;
  double sigma = (cartesian2[i]*invn - q*q)*invn;
  sigma = sqrt((sigma > 0.0 ? sigma : 0.0));

  q /= volumeCart;
```

```
  sigma /= volumeCart;

  results.data[i] = q;
  results.sigma[i] = sigma;
      }

      return results;
    });
```

Code 76: `setResultsGenerator` function call for `KERMA_TRACK_LENGTH` tally.

This function includes the results class initialization, labeling, and value assignment. These steps are described in the next section.

6.7.3 Results class

The template class `penred::measurements::results` and the associated `penred::measurements::measure` class were implemented to simplify future tally creation by automating the measurement addition and results generation process. Both implementations can be found in:

`src/kernel/includes/math_classes.hh`

These classes provide a container with useful functionality, such as automatic formatting, multi-dimensional handling, and uncertainty calculation.

Initialization

To initialize these objects, the number of bins for each dimension must be provided, along with the limits for each dimension. An example is shown in Code 77 for the `KERMA_TRACK_LENGTH` tally, specifically for the Cartesian mesh results. This consists of a 3D mesh along the three axes (X, Y, Z).

```
      penred::measurements::results<double, 3> results;
      results.initFromLists
  ({static_cast<unsigned long>(nbinsCart.x),
     static_cast<unsigned long>(nbinsCart.y),
     static_cast<unsigned long>(nbinsCart.z)},
    {penred::measurements::limitsType(minsCart.x, maxsCart.x),
     penred::measurements::limitsType(minsCart.y, maxsCart.y),
     penred::measurements::limitsType(minsCart.z, maxsCart.z)
    });
```

Code 77: Initialization of the `KERMA_TRACK_LENGTH` cartesian 3D mesh results object.

Initialization can be done using initialization lists, as in the previous example, or standard arrays. The signatures of both initialization functions are shown in Code 78.

```
      template<size_t dimInit>
      int init(const std::array<unsigned long, dimInit>& nBinsIn,
             const std::array<std::pair<double, double>, dimInit>& limitsIn)

      template<size_t binDims, size_t limitsDims>
      inline int initFromLists(const unsigned long(&nBinsIn)[binDims],
             const std::pair<double, double>(&limitsIn)[limitsDims])
```

Code 78: Results and measurement classes initialization methods.

Thus, two lists or arrays must be provided: the first with the number of bins per dimension, and the second with the limit pairs (bottom and top) for each dimension.

Labels

Both classes can store labels for values, uncertainties, and each dimension, along with a global description. These labels can be defined using:

- **description** member variable: A string storing a global description.
- **setDimHeader**: Takes the dimension index as its first argument and a string as its second argument for the corresponding header or label. This text will be added at the top of the dimension column when values are saved to disk. Additionally, the header can be read, encapsulating both values and documentation.
- **setValueHeader**: Same as **setDimHeader** but applied to values.
- **setSigmaHeader**: Same as **setDimHeader** but applied to uncertainties.

An example is shown in Code 79 for the Cartesian results within the `KERMA_TRACK_LENGTH` tally.

```
    results.description =
"PenRed: Cartesian kerma report";

    results.setDimHeader(0, "x (cm)");
    results.setDimHeader(1, "y (cm)");
    results.setDimHeader(2, "z (cm)");
    results.setValueHeader("Kerma (eV/g hist)");
```

Code 79: Initialization of `KERMA_TRACK_LENGTH` cartesian 3D mesh results labels.

Values and uncertainties

Once initialized and labeled, the values and their corresponding uncertainties must be set within the results object. It provides two accessible vectors, **data** and **sigma**, to store the data values and their uncertainties, respectively. How to fill these vectors depends on the tally and output structure. An example corresponding to the Cartesian results for the `KERMA_TRACK_LENGTH` tally is shown in Code **??**.

```
const size_t nBins = nbinsCart.x*nbinsCart.y*nbinsCart.z;
for(size_t i = 0; i < nBins; ++i){
  double q = cartesian[i]*invn;
  double sigma = (cartesian2[i]*invn - q*q)*invn;
  sigma = sqrt((sigma > 0.0 ? sigma : 0.0));

  q /= volumeCart;
  sigma /= volumeCart;

  results.data[i] = q;
  results.sigma[i] = sigma;
}
```

Code 80: Data and uncertainty initialization for the `KERMA_TRACK_LENGTH` Cartesian 3D mesh results.

6.8 Automatic Scoring

Instead of manually creating scoring containers, performing boundary checks, and building **results** objects for output, these steps can be handled automatically by the `penred::measurements::measure` class. This **measurement** object is initialized and labeled using the same functions as the

results class (Section 6.7.3). Once initialized, results can be accumulated using the **add** method (Code 81). This method takes an array with the position in each dimension, the value to add, and the history number.

```
void add(const std::array<double, dim>& pos,
         const type& value,
         const unsigned long long hist)
```

Code 81: Add method for the **measurement** class.

An example is shown in Code 83 for the `DETECTION_SPATIAL_DISTRIB` tally, where the *results* variable is a `penred::measurements::measure<double,4>` instance. This 4D scoring mesh has dimensions of (energy, x, y, z). The particle's weight is accumulated in this case as the value.

```
void pen_DetectionSpatialDistrib::tally_interfCross(const unsigned long long nhist,
                 const unsigned kdet,
                 const pen_KPAR kpar,
                 const pen_particleState& state){

  if(kdet == idet && kpar == iPar){
    results.add({state.E, state.X, state.Y, state.Z}, state.WGHT, nhist);
  }
}
```

Code 82: Example of the **add** method usage for the **measurement** class.

The history number is used to avoid updating all temporary buffer bins used for uncertainty calculation. Instead, only those bins where a new history scores are updated. Due to this approach, a **flush** method is provided to retrieve the final or partial results. Therefore, the tally's **flush** method can be defined by calling it, as is done for the `DETECTION_SPATIAL_DISTRIB` tally (Code **??**).

```
inline void flush(){ results.flush(); }
```

Code 83: Example of the **flush** method usage for the **measurement** class.

Adding results

To add the results from different threads, the **measurement** class implements an **add** method overload that accepts an instance of the same type. Therefore, the **sumTally** method can be simplified, as shown in Code 84.

```
int pen_DetectionSpatialDistrib::sumTally(const pen_DetectionSpatialDistrib& tally){
  return results.add(tally.results);
}
```

Code 84: Example of the **add** method usage to accumulate different **measurement** instances.

Results saving

For saving results, the **print** method can be used to create result files, as shown in Code 85.

```
void pen_DetectionSpatialDistrib::saveData(const unsigned long long nhist) const{

  std::string filename("spatial-detection-");
```

```
  filename += std::to_string(idet);
  filename += ".dat";

  FILE* out = nullptr;
  out = fopen(filename.c_str(), "w");

  results.print(out, nhist, 2, printCoord, printBins);
}
```

Code 85: Example of the **print** method usage for the **measurement** class.

The print method signature is shown in Code 86, where the following parameters must be specified:

- **fout**: Output file pointer.
- **nhists**: Number of simulation histories. It is used to normalize the results.
- **nSigma**: Number of standard deviations to report.
- **printCoordinates**: If enabled, coordinates for each bin are calculated according to the initialization limits and reported in the results file.
- **printBinNumber**: If enabled, bin numbers are reported in the results file.
- **printOnlyEffective**: If enabled, only dimensions with more than one bin are reported.

```
void print(FILE* fout,
       const unsigned long long nhists,
       const unsigned nSigma,
       const bool printCoordinates,
       const bool printBinNumber,
       const bool printOnlyEffective = false) const
```

Code 86: **print** method signature of the **measurement** class.

Results generation

The **measurement** object allows the creation of a **results** object using its internal data and the **results** method. This can be used to define the output generation functions, as shown in Code 87.

```
  pen_DetectionSpatialDistrib() : pen_genericTally(USE_INTERFCROSS |
                USE_MOVE2GEO)
  {
    //Register results functions
    setResultsGenerator<0>
      ([this](const unsigned long long nhists) -> penred::measurements::
    results<double,4>{
      penred::measurements::results<double, 4> generated;
     results.results(nhists, generated);
     return generated;
    });
  }
```

Code 87: Results object generation using the **measurement** class.

7 Source samplers

Source samplers are used to sample the particle states to be simulated. Samplers are classified into five categories: `spatial`, `direction`, `energy`, `time`, and `specific`. The purpose of each category is explained below:

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

Thus, we will consider **Spatial**, **Direction**, **Energy**, and **Time** samplers as generic samplers, as they can be used with any particle state.

The following sections describe how to implement a sampler of each type. The main differences between sampler types are the state variables they calculate. However, all samplers share a common function that must be implemented by the developer: the virtual *configure* method. Its signature is sampler-dependent, but all *configure* methods must return an integer value. A return value of 0 indicates success, while a nonzero value indicates failure.

Additionally, samplers may require geometry information to perform their calculations. For this reason, when a geometry is assigned to the samplers, the virtual method *updateGeometry* (Code 88) is triggered, allowing the sampler to use and store any information provided by the geometry. This method can be used in all sampler types and must be overridden by the developer to utilize geometry information if required. Note that during the execution of *configure*, the geometry is not accessible and must be accessed within the *updateGeometry* method.

```

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

Code 88: Sampler `updateGeometry` virtual function.

7.1 Register

To register a sampler, the developer must use the provided macros to declare and register it. The "declare sampler" macro is shown in Code 89 and takes one or two arguments depending on the sampler type:

- **Generic** (DECLARE_SAMPLER): Takes a single argument corresponding to the class name.
- **Specific** (DECLARE_SPECIFIC_SAMPLER): Takes the class name and particle state as first and second arguments respectively.

```

DECLARE_SAMPLER(Class)
DECLARE_SPECIFIC_SAMPLER(Class, State)
```

Code 89: Sampler declaration macro.

These declaration macros must be used inside the sampler class definition, immediately after the class name, as shown in Code 90 for the spatial box sampler. The same procedure is used to declare all samplers, regardless of their type.

```

class box_spatialSampling : public abc_spatialSampler {

  DECLARE_SAMPLER(box_spatialSampling)

   .
   .
   .
}
```

Code 90: Sampler declaration macro example.

On the other hand, two "register sampler" macros are provided, as shown in Code 91.

```

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

Code 91: Sampler registration macros.

Only specific samplers must use the **REGISTER_SPECIFIC_SAMPLER** macro, while the other types must use the **REGISTER_SAMPLER** macro. The *Class* parameter corresponds to the sampler class name, the *State* parameter specifies the particle state class compatible with the specific sampler, and the *ID* parameter provides a unique identifier for the registered sampler. The corresponding register macro must be called in the source file, outside any function scope. Examples of each macro usage are shown in Code 92, where *box_spatialSampling* and *gammaPolarised_specificSampler* are used for generic and specific sampler registration, respectively.

```

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

Code 92: Sampler register macros.

Note that a specific sampler does not necessarily require a non-generic particle state. In fact, a specific sampler can be used to sample basic particle states. This is the case for the phase space file sampler, whose registration macro is shown in Code 93.

```

REGISTER_SPECIFIC_SAMPLER(psf_specificSampler,pen_particleState, PSF)
```

Code 93: Sampler register macro with generic particle state.

7.2 Spatial

Custom `spatial` samplers must inherit from the base class `abc_spatialSampler` using:

```
public abc_spatialSampler
```

This base class is located in the file:

`src/particleGen/includes/pen_samplers.hh`

Several examples of `spatial` sampler implementations can be found in:
`src/particleGen/spatial`

The samplers are intended to provide an 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 vector, R is a rotation matrix, and X is the initial sampled position. Both T and R are variables provided in the base class `abc_spatialSampler`, corresponding to the variables:

`double translation[3];`

and

`double rotation[9];`

respectively. The *translation* variable can be accessed directly in derived classes and is intended to be filled in the *configure* function. The *rotation* variable should also be filled during the *configure* execution, but using the public method *setRotationZYZ* (Code 94), which takes three Euler angles as arguments to calculate the final rotation matrix. Note that the Euler angles follow the same convention as the PENELOPE package, i.e., rotations are performed using the Z, Y, Z axes. Rotation is optional; if no rotation matrix is 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 94: Set rotation method.

There are no more common variables to fill in the spatial sampler; the remaining variables are specific to the implemented sampler. Regarding functions, the only two required methods to implement a `spatial` sampler are *configure* and *geoSampling*, whose signatures are shown in Code 95.

```

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

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

Code 95: Spatial sampler mandatory functions.

First, the *configure* method takes a `pen_parserSection` (Section 3) as its first argument, which contains all the data specified by the user to configure the sampler. Additionally, the *verbose* parameter should be used to control the volume of printed information. An example is shown in Code 96, which corresponds to the box spatial sampler. As seen in this configuration function, it 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 96: `box_spatialSampling` configure function.

Second, the *geoSampling* method receives an array of three doubles, which must be filled with the sampled x, y, z position (vector X from Equation 3). A random number generator is provided via the *random* parameter. Again, the box spatial sampler is used as an example, and its corresponding *geoSampling* method is shown in Code 97.

```

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 97: Spatial sampler *geoSampling* function.

Note that the *geoSampling* method is defined as constant. Therefore, this method cannot change the sampler state. Modifications are only allowed during 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 registration steps described in Section 7.1, the developer must include the corresponding header and source file names in the files:

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

and

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

respectively. Additionally, in the header file, the sampler class name must be added to the tuple type *typesGenericSpatial* (Code 98). Note that the sampler must be added to both definitions (with and without DICOM enabled) to be available under all compilation conditions.

```
#ifdef _PEN_USE_DICOM_
    using typesGenericSpatial = std::tuple<box_spatialSampling,
              point_spatialSampling,
              image_spatialSampling,
              cylinder_spatialSampling,
              measure3D_spatialSampling,
              measure2D_spatialSampling,
              measure1D_spatialSampling>;
```

```
#else
    using typesGenericSpatial = std::tuple<box_spatialSampling,
                point_spatialSampling,
                cylinder_spatialSampling,
                measure3D_spatialSampling,
                measure2D_spatialSampling,
                measure1D_spatialSampling>;
#endif
```

Code 98: Definition of `typesGenericSpatial` type.

7.3 Direction

`Direction` samplers must inherit from the base class `abc_directionSampler` using:

`public abc_directionSampler`

The base class is located in the file:

`src/particleGen/includes/pen_samplers.hh`

Several examples of `direction` sampler implementations can be found in:

`src/particleGen/direction`

This kind of sampler is intended to provide the initial **normalized** direction vector (U, V, W) of the particle. Unlike spatial samplers, direction samplers do not require filling any variables in the base class. Therefore, a direction sampler only needs to implement the *configure* method and the sampling function, named *directionSampling*. Their signatures are shown in Code 99.

```

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

Code 99: Direction sampler mandatory functions.

First, the *configure* method takes a `pen_parserSection` (Section 3) as its first argument, which contains all the data specified by the user to configure the sampler. Additionally, the *verbose* parameter should be used to control the volume of printed information. An example is shown in Code 100, which corresponds to the conic direction sampler.

```

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 100: Configure implementation of the conic direction sampler.

Second, the *directionSampling* method receives an array of three doubles, which must be filled with the **normalized** sampled direction (U, V, W). A random number generator is provided via the *random* parameter. Code 101 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 101: Conic direction sampler *directionSampling* implementation.

Note that the *directionSampling* method is defined as constant. Therefore, this method cannot change the sampler state. Modifications are only allowed during 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 registration steps described in Section 7.1, the developer must include the corresponding header and source file names in the files:

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

and

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

respectively. Additionally, in the header file, the sampler class name must be added to the tuple type *typesGenericDirection* (Code 102).

```
    using typesGenericDirection = std::tuple<sphereSection_directionSampling,
                                        cone_directionSampling>;
```

Code 102: Definition of `typesGenericDirection` type.

7.4 Energy

Custom `energy` samplers must inherit from the base class `abc_energySampler` using:

`public abc_energySampler`

The base class is located in the file:

`src/particleGen/includes/pen_samplers.hh`

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. Like `direction` samplers, **energy** samplers do not need to fill any base class member variables but must implement two functions: the *configure* method and the sampling method named *energySampling*. Both function declarations are shown in Code 103.

```cpp

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 103: Energy sampler mandatory functions.

The *configure* method takes a **pen_parserSection** (parameter *config*) to obtain the configuration data specified by the user and the *verbose* parameter to control the print verbosity, just like previous samplers. However, it also takes an extra parameter named *Emax*. This is an output parameter and must be filled with the maximum energy that the sampler can return in any call to the *energySampling* method. It is important to fill this value correctly, as the maximum energy from all sources involved in the simulation will be used to construct the required energy grids for the simulation. An example of the *configure* function is shown in Code 104, which corresponds to the **monoenergetic** energy sampler.

```cpp

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 104: Monoenergetic sampler configuration function.

The *energySampling* method takes two parameters. The first one, named *Energy*, must be filled with the sampled energy in eV. It also takes an instance of the **pen_rand** class (parameter *random*) to randomize the energy sampling. Code 105 shows the implementation of the *energySampling* method for the **monoenergetic** sampler. As it is a monoenergetic source, the random generator is not needed.

```cpp

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

```
  energy = E;
}
```

Code 105: Monoenergetic sampler *energySampling* implementation.

Note that the *energySampling* method is defined as constant. Therefore, this method cannot change the sampler state. Modifications are only allowed during 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 registration steps described in Section 7.1, the developer must include the corresponding header and source file names in the files:

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

and

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

respectively. Additionally, in the header file, the sampler class name must be added to the tuple type *typesGenericEnergy* (Code 106).

```
  using typesGenericEnergy = std::tuple<intervals_energySampling,
                                        monoenergetic,
                                        fileSpectrum_energySampling>;
```

Code 106: Definition of `typesGenericEnergy` type.

7.5 Time

Custom `time` samplers must inherit from the base class `abc_timeSampler` using:

`public abc_timeSampler`

The base class is located in the file:

`src/particleGen/includes/pen_samplers.hh`

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 an initial particle time in seconds. If no time sampler is specified, the particle time will be set to 0 during state sampling. Also, note that using a time sampler does not automatically enable particle time-of-flight calculations. This must be enabled explicitly in the configuration file. For example, in the penRed main program, this must be specified as:

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

where *source1* is the source name and may vary depending on the simulation.

Regarding implementation, time samplers do not have any base class member variables to fill, leaving only two required functions to implement. As in other cases, these functions are the *configure* method and the sampling method, named *timeSampling*. Both declarations are shown in Code 107.

```

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

Code 107: Time sampler mandatory functions.

The *configure* method takes a `pen_parserSection` as its first argument, which contains all the data specified by the user to configure the sampler. Additionally, the *verbose* parameter should be used to control the volume of printed information. An example is shown in Code 108, 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 108: Decay time sampler configuration function.

Regarding the sampling function, the first parameter of the *timeSampling* method, named *time*, must be filled with the sampled time in seconds. An instance of the `pen_rand` class is also provided (parameter *random*) to enable random time sampling. An example is shown in Code 109.

```

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

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

Code 109: Decay time sampler *timeSampling* function.

Note that the *timeSampling* method is defined as constant. Therefore, this method cannot change the sampler state. Modifications are only allowed during 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 registration steps described in Section 7.1, the developer must include the corresponding header and source file names in the files:

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

and

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

respectively. Additionally, in the header file, the sampler class name must be added to the tuple type *typesGenericTime* (Code 110).

```
    using typesGenericTime = std::tuple<decay_timeSampling>;
```

Code 110: Definition of `typesGenericTime` type.

7.6 Specific

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

```
public abc_specificSampler<particleState>
```

where *particleState* specifies the particle state class. The base class is located in the file:

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

Several examples of `specific` sampler implementations can be found in:

```
src/particleGen/specific
```

Specific samplers are more versatile and complex than the previous types. The first difference is that specific samplers can handle a **specific particle state**, specified by the template parameter *particleState*. Additionally, 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 is used by the phase space file sampler, whose class declaration is shown in Code 111.

```

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

Code 111: Phase space file sampler class declaration.

7.6.1 Sampler function

Another key difference is the sampler function, named *sample*, whose declaration is shown in Code 112.

```

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

Code 112: Specific sampler *sample* function.

Note that the *sample* function is not declared as a constant method. Therefore, `specific` samplers are allowed to change their state during sampling calls. However, the developer must ensure that the *sample* function is thread-safe to utilize multi-threading capabilities. To facilitate this, each simulation thread uses its own private instance of the specific sampler, which is not shared with other threads. Additionally, if a shared resource must be used by several threads, the base class `abc_specificSampler` provides a method to obtain a thread number identifier, named *getThread* (Code 113).

```

```

```
  inline unsigned getThread() const {return nthread;}
```
Code 113: Specific sampler *getThread* method.

The *sample* function is more complex than in previous cases. As shown in Code 112, the first argument is the entire particle state to be filled. The second argument, of type `pen_KPAR`, should be filled with the sampled particle type, allowing a specific sampler to produce different particle types (particle type indexes can be found in the penRed user manual). The parameter *dhist* must be filled with the history increment. Generic samplers typically set this to 1, meaning each sampled particle corresponds to a new history. However, specific samplers can change this, sampling several particles in the same history or skipping more than one history between consecutive *sample* calls. Finally, the *random* parameter provides an instance of the `pen_rand` class for generating random numbers.

7.6.2 Sampling generic state

Although the sampler can change the entire state, it can delegate part or all of the generic state sampling to generic samplers (i.e., position to spatial samplers, direction to direction samplers, energy to energy samplers, and time to time samplers).

If the developer wants to delegate the sampling of some parts of the generic state, this must be indicated in the constructor via the specific sampler base class constructor (Code 114) using a set of predefined flags (Code 115). The procedure will be exemplified later.

```

  abc_specificSampler(__usedSamp usedSamplingsIn) :
                usedSamplings(usedSamplingsIn),
                pSpatial(nullptr),
                pDirection(nullptr),
                pEnergy(nullptr),
                pTime(nullptr)
  {}
```
Code 114: 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 115: Generic sampling usage flags for specific samplers.

Flags can be combined using the | operator, like:
`(USE_SPATIAL | USE_DIRECTION)`

The flag descriptions are as follows:

- **USE_SPATIAL**: A generic `spatial` sampler must be used.
- **USE_DIRECTION**: A generic `direction` sampler is required.
- **USE_ENERGY**: A generic `energy` sampler is mandatory.
- **USE_TIME**: Requires the usage of a generic `time` sampler.

- **USE_GENERIC**: Indicates that a `spatial`, `direction`, and `energy` generic samplers are required, but the `time` sampler is optional. With this flag, the generic part of the particle state will be sampled automatically as if no specific sampler were defined. Then, the *sample* method of the specific sampler will be called and can overwrite any part of the previous sampling.

- **USE_NONE**: Indicates that no generic sampler is required for this specific sampler.

Note 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, their configuration will be handled automatically. Therefore, using the flags **USE_SPATIAL**, **USE_DIRECTION**, and **USE_ENERGY** does not yield the same result as using the **USE_GENERIC** flag. In the latter case, the generic samplers are handled automatically by the framework, and the specific sampler function will receive the generic part of the *state* parameter already sampled.

Also, note that the flags specify which generic samplers are mandatory. If a mandatory generic sampler is not provided by the user in the configuration file, the configuration will return an error and execution will stop. However, if a non-mandatory sampler is provided by the user, it could still be used by the specific sampler depending on its implementation, though a warning will be printed to indicate that the corresponding generic sampler might be ignored.

The following examples illustrate two specific samplers. The first (Code 116) shows the specific sampler for polarized photons, which delegates all generic sampling 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 116: Gamma polarised photons specific sampler class header.

Thus, its *sample* function does not need to handle the generic state part. As shown in Code 117, the *sample* method only sets the polarization-related variables specific to the `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 117: Gamma polarised photons *sample* function.

The next example handles the generic state sampling completely and corresponds to the phase space file specific sampler. Thus, no generic sampler is required, specified using the **USE_NONE** flag (Code 118).

```

  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 118: Phase space file constructor.

To access the assigned `spatial`, `direction`, `energy`, or `time` samplers, the `abc_specificSampler` provides a set of methods described in Section 7.6.6.

7.6.3 Configuration

The `specific` sampler configuration virtual method is defined in Code 119.

```

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

```

Code 119: Specific sampler configuration method.

Like previous samplers, this configuration method takes a `pen_parserSection` instance named *config*, which provides all user-specified configuration parameters. The *verbose* parameter controls the verbosity level. Like `energy` samplers, the *Emax* parameter must be filled with the maximum energy the sampler can return. Note that this parameter does not need to be filled if a complete generic sampler is used (enabled via the **USE_GENERIC** flag in the constructor). Finally, the *nthreads* parameter provides the number of threads to be used in the simulation.

When the configuration function is called, all possible generic samplers (*spatial*, *direction*, *energy*, and *time*) are already stored in their corresponding pointers, or are null if not configured by the user. Therefore, these pointers can be accessed during the configuration call using the functions described in Section 7.6.6. However, note that the geometry cannot be accessed during the *configure* method call (the *geo* function will return a null pointer). Geometry-related calculations must be done in the *updateGeometry* method (Code 88), which is called when a geometry is assigned to the samplers.

7.6.4 Resume a simulation

Allowing the specific sampler state to change during the sampling method means that sampling can depend on the number of previous *sample* calls (e.g., if states are read from a file). This can be problematic if the simulation is resumed from a dump file, as the same information might be read twice.

To handle this situation, the specific sampler base class provides a virtual method named *skip* (Code 120).

```

virtual void skip(const unsigned long long /*dhists*/){}
```

Code 120: *Skip* virtual method of the `abc_specificSampler` class.

The *skip* method takes the number of histories to skip (parameter *dhists*) as its only argument. This function only needs to be implemented if required by the sampler, typically when the state changes during the *sample* call (e.g., the phase space file sampler reads from a file). The gamma polarized 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 121). Currently, this method has no use but is included to allow changing generic samplers during the simulation in future implementations. Therefore, this method **must not** be implemented.

```

virtual void updateSamplers(){}
```

Code 121: 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 auxiliary methods to access information for the sampling process (Code 122).

```

  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 122: Auxiliary methods provided in the `abc_specificSampler` class.

Their descriptions are as follows:

- **spatial**: Returns a pointer to the assigned `spatial` sampler. Returns a null pointer if not assigned.
- **direction**: Returns a pointer to the assigned `direction` sampler. Returns a null pointer if not assigned.
- **energy**: Returns a pointer to the assigned `energy` sampler. Returns a null pointer if not assigned.

- **time**: Returns a pointer to the assigned `time` sampler. Returns a null pointer if not assigned.
- **geo**: Returns a pointer to the assigned geometry. Returns a null pointer if not assigned.
- **getThread**: Returns a numeric identifier for the thread that owns the specific sampler instance.

Note that to run a generic sampling with the provided pointers, the public *sample* method of the generic sampler must be called, not the overridden method used to implement it. For example, to use a `spatial` sampler, call its *sample* method, not the *geoSampling* method.

7.6.7 Including files

To use a developed specific sampler in the penRed environment, in addition to the sampler registration steps described in Section 7.1, the developer must include the corresponding header and source file names in the files:

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

and

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

respectively. Additionally, in the header file, the sampler class name must be added to the tuple type *typesSpecificCommonState* (Code 123). Note that the sampler must be added to both definitions (with and without DICOM enabled) to be available under all compilation conditions.

```
#ifdef _PEN_USE_DICOM_
    using typesSpecificCommonState = std::tuple<random_specificSampler,
            psf_specificSampler,
            ct_specificSampler,
            pennuc_specificSampler,
            brachy_specificSampler,
            psfMemory_specificSampler>;
#else
    using typesSpecificCommonState = std::tuple<random_specificSampler,
            psf_specificSampler,
            ct_specificSampler,
            pennuc_specificSampler,
            psfMemory_specificSampler>;
#endif
```

Code 123: Definition of `typesGenericTime` type.

8 Variance reduction (VR)

Variance reduction techniques (VR) are used to improve the efficiency of estimating simulation quantities. VR modules inherit from the class:
`pen_genericVR`
whose source can be found in:

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

This class is a template interface that takes a particle state type as its template argument. 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 using:

```
public pen_genericVR<pen_particleState>
```

Otherwise, a different particle state can be specified to limit the VR module's usability. For example, the x-ray splitting module (Code 124) is limited to photons with the state `pen_state_gPol`.

```

class pen_VRxraysplitting : public pen_genericVR<pen_state_gPol>
```

Code 124: *pen_VRxraysplitting* class declaration.

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

8.1 Register

Implemented VR modules must be registered using the *DECLARE_VR* and *REGISTER_VR* macros (Code 125). These macros take as arguments the class name of the 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 125: VR register macros.

The first macro must be placed inside the class declaration, immediately after the class name, as shown in Code 126, which corresponds to the *pen_VRRussianRoulette* class definition.

```

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

Code 126: VR declare macro.

The second macro must be placed in the source file, as shown in Code 127.

```

REGISTER_VR(pen_VRRussianRoulette, pen_particleState, RUSSIAN_ROULETTE)
```

Code 127: VR register macro.

The source file inclusion depends on the specified particle state. Modules that use the generic particle state (`pen_particleState`) are classified as generic VR modules. If another particle state is used, the module is considered specific.

8.1.1 Generic modules files inclusion

To use a developed generic VR module in the penRed environment, in addition to the VR registration steps described in Section 8.1, the developer must include the corresponding header and source file names in the files:

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

and

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

respectively. Additionally, in the header file, the VR class name must be added to the tuple type *typesGenericVR* (Code 128).

```
using typesGenericVR = std::tuple<pen_VRsplitting,
        pen_VRRussianRoulette,
        pen_VRradialSplitting>;
```

Code 128: Definition of `typesGenericVR` type.

8.1.2 Specific modules files inclusion

To use a developed specific VR module in the penRed environment, in addition to the VR registration steps described in Section 8.1, the developer must include the corresponding header and source file names in the files:

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

and

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

respectively. Additionally, in the header file, the VR class name must be added to the tuple type *typesSpecificVR* (Code 129).

```
using typesSpecificVR = std::tuple<pen_VRxraysplitting>;
```

Code 129: Definition of `typesSpecificVR` type.

8.2 VR callbacks

Like tallies (Section 6), VR modules run at specific points during the simulation execution. These points can be enabled or disabled for each implemented VR module depending on its needs. Each point is handled by a specific callback, described in this section. All callbacks receive the variables shown in Code 130.

```

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 130: Common VR callbacks parameters.

First, *nhist* specifies the history number. Second, *kpar* specifies the particle type to which the VR technique will be applied. Then, *state* stores the particle state. Note that this state will usually change during the call and is therefore 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 clone particles (e.g., techniques based on particle splitting), the *stack* parameter provides storage for generated particle states. The *stack* size is limited to *constants::NMS* states. However, since generated particles will be added to the secondary particle stack (which is not directly accessible), the developer must consider the *created* and *available* parameters. The first (*created*) stores the number of states already stored in the provided *stack* array, which may have been filled by a previous VR callback. Therefore, when a new state is stored in the *stack*, the value of *created* must be incremented. 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 previously created state could be overwritten. Finally, the parameter *available* stores how many free spaces are available in the secondary particle stack. This value is constant, as the secondary particle stack does not change until all VR callbacks have been called. The remaining free space can be calculated as:

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

Note that the *created* parameter may have a non-zero value when the callback is called, as this parameter is passed to subsequent VR module callbacks. After all callbacks have executed, the *stack* parameter will contain all 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 that provides a random number generator.

All available callbacks are listed below:

- **vr_matChange**: Called when the material where the particle is located changes. An extra argument, *prevMat*, stores the previous material.

```

    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 131: VR material changed callback.

- **vr_interfCross**: Called when the particle crosses an interface. An extra argument, *kdet*, stores the detector index where the particle is located.

```cpp

    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 132: VR interface crossed callback.

- **vr_particleStack**: Called when a particle state is extracted from the secondary stack. An extra argument, *kdet*, stores the detector index where the particle is located.

```cpp

    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 133: VR secondary particle extracted from stack callback.

All callbacks are defined as constant methods, so changing the module state is not possible during the callback call.

8.3 Callbacks to trigger

Each VR module does not need to implement all available callbacks. Only the necessary methods should be implemented. To specify which methods a VR module will use, the corresponding flags must be passed as an argument to the interface class constructor (`pen_genericVR`). These flags are:

VR_USE_PARTICLESTACK

VR_USE_MATCHANGE

VR_USE_INTERFCROSS

For example, Code 134 shows the constructor of the splitting VR module, which only requires the interface cross callback.

```cpp

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

Code 134: VR splitting module constructor.

Flags can be combined using the binary OR operator (|). For example, to enable the *particleStack* and *matchange* flags, the constructor call must be made as shown in Code 135.

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

Code 135: 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 136). It takes as arguments a `pen_parserSection` instance named *config*, which contains all the necessary user-specified configuration information for the VR module; the geometry information via the parameter *geometry*; and a *verbose* parameter to control the print verbosity.

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

Code 136: 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